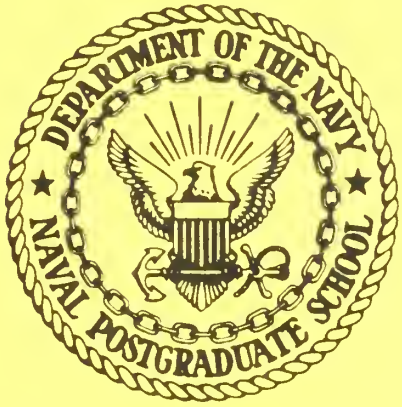
NPS52-84-005

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THE IMPLEMENTATION OF A MULTI-BACKEND DATABASE SYSTEM (MDBS): PART IV - THE REVISED CONCURRENCY CONTROL AND DIRECTORY MANAGEMENT PROCESSES AND THE REVISED DEFINITIONS OF INTER-PROCESS AND INTER-COMPUTER MESSAGES

Steven A. Demurjian, David K. Hsiao
Douglas S. Kerr and Ali Orooji

February 1984

Prepared for:

Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Commodore R. H. Shumaker
Superintendent

D. A. Schrady
Provost

The work reported herein was supported by Contract N00014-84-WR-24058 from the Office of Naval Research.

This report was prepared by:

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NPS52-84-005

2. GOVT ACCESSION NO.:

3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): The Implementation of a Multi-backend Database System (MDBS): Part IV - The Revised Concurrency Control and Directory Management Processes and the Revised Definitions of Inter-Process and Inter-Computer Messages

5. TYPE OF REPORT & PERIOD COVERED:

6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s): Steven A. Demurjian, David K. Hsiao, Douglas S. Kerr and Ali Orooji

8. CONTRACT OR GRANT NUMBER(s): N00014-84-WR-24058

9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93943

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:

11. CONTROLLING OFFICE NAME AND ADDRESS: Chief of Naval Research, Arlington, VA 22217

12. REPORT DATE: February 1984

13. NUMBER OF PAGES: 112

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):

15. SECURITY CLASS. (of this report): Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report):

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report):

18. SUPPLEMENTARY NOTES:

19. KEY WORDS (Continue on reverse side if necessary and identify by block number):
Database systems, concurrency control, directory management, message passing, multi-backends, attribute search, descriptor search, cluster search, address generation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number):
The multi-backend database system (MDBS) uses one minicomputer as the master or controller, and a varying number of minicomputers and their disks as slaves or backends. MDBS is primarily designed to provide for database growth and performance enhancement by the addition of identical backends. No special hardware is required. The backends are configured in a parallel manner. A new backend may be added by replicating the existing software on the new back-end. No new programming or reprogramming is required.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

A prototype MDBS is being completed in order to carry out the design verification and performance evaluation. This report is the fourth in a series which describes the MDBS implementation. In the report, an overview of MDBS and its process structure is first given. The processes in the MDBS controller (request preparation, insert information generation and post processing) and the processes in the MDBS backends (directory management, record processing and concurrency control) have been described in the previous reports. In this report, the changes made in the concurrency control and directory management processes are then discussed.

The concurrency control process, formerly used to control access to just user data, is modified to control access to directory data as well. The directory management process is also modified to improve the execution of update requests. Finally, directory management is modified for the storage of directory data on the secondary storage.

Next, the report describes the revised definitions of inter-process messages (messages between processes within a minicomputer) and inter-computer messages (messages between processes in different minicomputers). A detailed description of the sequences of actions for directory processing is also given.

Finally, we conclude this series of reports dealing with the implementation of MDBS. We also review the next phase of development, which includes a hardware reconfiguration and expansion, a security mechanism, language interfaces to support the relational and hierarchical data manipulation languages, and the performance evaluation of MDBS.

The appendices contain the detailed design for the concurrency control process for directory data and the revisions to directory management due to the storage of directory data on the secondary storage.

TABLE OF CONTENTS

PREFACE

This work is supported by Contract N00014-84-WR-24058 from the Office of Naval Research to Dr. David K. Hsiao and conducted in the Laboratory for Database Systems Research in the Department of Computer Science at the Naval Postgraduate School (NPS). The Laboratory for Database Systems Research was initially funded by the Digital Equipment Corporation (DEC), Office of Naval Research (ONR) and the Ohio State University (OSU) and consists of the staff, graduate students, undergraduate students, visiting scholars and faculty for conducting research and teaching in database systems. In July 1983 the Laboratory was transferred to NPS and is now supported by ONR and NPS. At that time the VAX-11/780 was given to OSU. Two PDP-11/44s with associated disk and tape drives, the three intercomputer communication devices (PCL-11Bs), five terminals, and one printer were transferred to NPS and linked to a VAX-11/780 at NPS. The work described in this technical report was started at OSU and has been completed at NPS.

We would like to thank all those who have helped with the MDBS project. In particular, the MDBS design and analysis were developed by Jai Menon. (Now, Dr. Jai Menon of IBM Research Laboratory, San Jose, California.) He has also provided much help in the detailed designs. A visiting scholar at OSU, Xing-Gui He, has been involved with the MDBS project. Several undergraduate students at OSU have also been involved with the project: Raymond Browder, Chris Jeschke, Jim McKenna, and Joe Stuber. Several graduate students, visiting scholars and undergraduate students at OSU have provided much help in the detailed design and coding: Steven Barth, Julie Bendig, Abdulrahim Beram, Richard Boyne, Patti Dock, Masanobu Higashida, Jim Kiper, Drew Logan, William Mielke, Tamer Ozsu, Zong-Zhi Shi, and Paula Strawser. Jose Alegria, Tom Bodnovich and David Brown have contributed background material which was necessary for making our design decisions. We would also like to thank the laboratory staff and other operators at OSU who have provided us with system support: Bill Donovan, Doug Karl, Paul Placeway, Steve Romig, Jim Skon, Dennis Slaggy, Mark Verber, and Geoff Wyant.

At NPS, we have received strong support form the professional staff of the Department of Computer Science. In particular, we would like to thank Albert Wong for his diligent work on VAX and PDP-11 software and Mike Williams and Walt Landaker for their good work on hardware installation. Finally we

would like to thank the School and the Department for providing an ideal environment for database system research.

LIST OF FIGURES

1. INTRODUCTION

This report is the fourth in a series describing the implementation of MDBS, a multi-backend database system [Kerr82, He82, Boyn83]. The original design was given in [Hsia81a, Hsia81b]. It is assumed that the reader is already familiar with these earlier reports. We will, however, give a very brief review of the MDBS design.

An overview of MDBS hardware organization is shown in Figure 1. MDBS is connected to a host computer through the controller. The controller and backends are, in turn, connected by a broadcast bus. The controller receives requests from a host computer. It then broadcasts each request to all backends at the same time over the bus. Since the database is distributed across the backends, a request can be executed by all backends in parallel.

To manage the database (often referred to as user data), MDBS uses directory data. Directory data in MDBS corresponds to attributes, descriptors, and clusters. An attribute is used to represent a category of the user data; e.g., SALARY is an attribute that corresponds to actual salaries stored in the database. A descriptor is used to describe a range of values that an attribute can have; e.g., (10001 <= SALARY <= 15000) is a possible descriptor for the attribute SALARY. The descriptors that are defined for an attribute, e.g., salary ranges, are mutually exclusive. Now the notion of a cluster can be defined. A cluster is a group of records such that every record in the cluster satisfies the same set of descriptors. For example, all records with SALARY between $10,001 and $15,000 may form one cluster whose descriptor is the one given above. In this case, the cluster satisfies the set of a single descriptor. In reality, a cluster tends to satisfy a set of multiple descriptors.

The process structure of MDBS is shown in Figure 2. A major design goal for MDBS was to minimize the work done by the controller and to maximize the work done by the backends. The controller must, however, perform some functions. It must first prepare a request for execution by the backends. This function is performed by request preparation. The controller must also coordinate responses from the backends. This function is performed by post processing. In addition, for consistency reasons, certain functions required for record insertion must also be performed in the controller. These functions are performed by insert information generation.

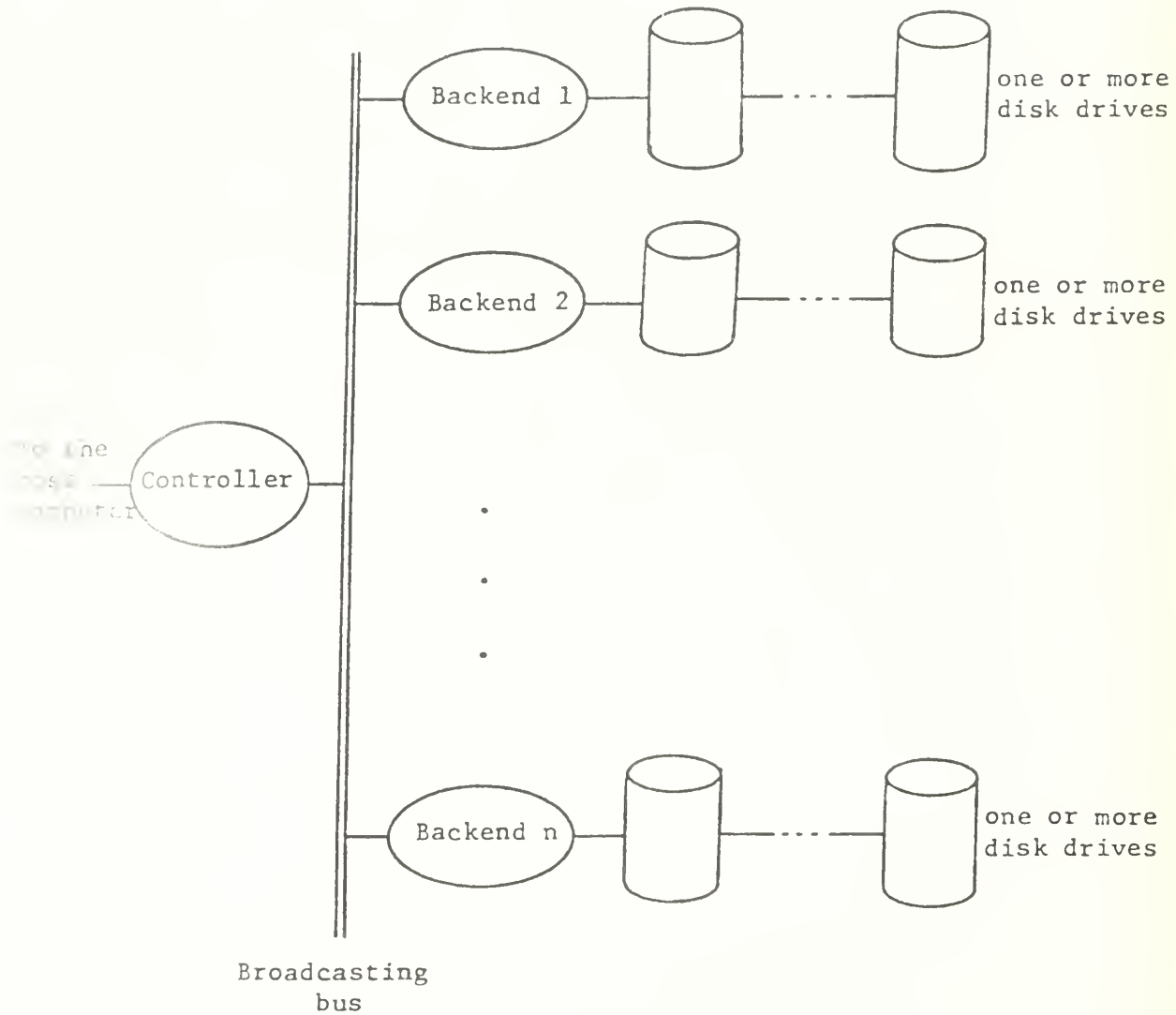


Figure 1. The MDBS Hardware Organization

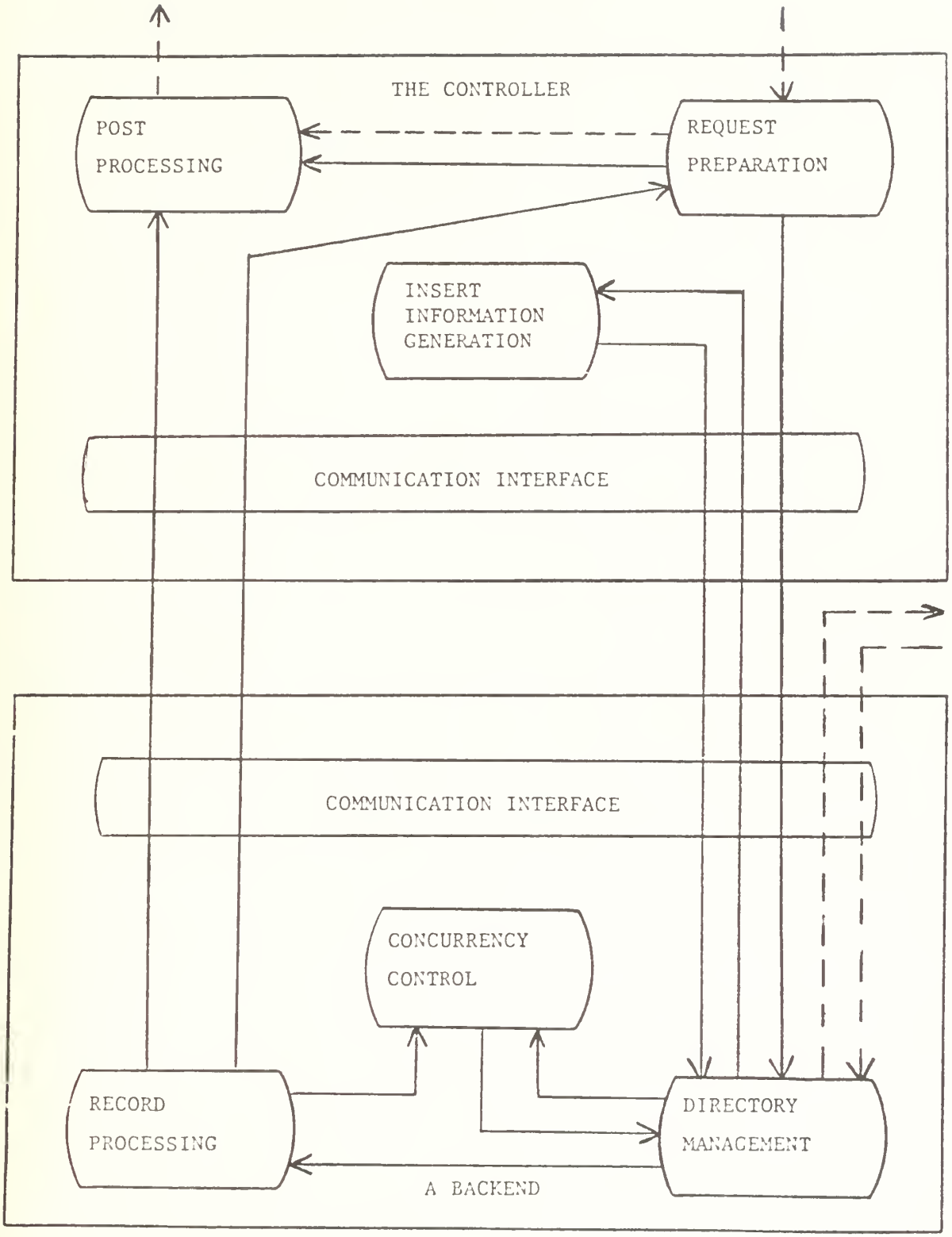


Figure 2. The MDBS Process Structure

As much work as possible has been given to the backends, this work consists of three categories of functions: directory management, concurrency control and record processing. The directory management functions are used to determine the addresses of the records required to process a particular request. The concurrency control function allows concurrent access to the database by different requests. The record processing functions perform the actual data retrieval and storage as well as the processing required on any particular record (e.g., the computation of a maximum value).

1.1. The Implementation Strategy

In this section we provide the reader with a brief review of the MDBS implementation strategy. Recall that the implementation strategy involved the development of MDBS in seven versions, labeled version A to version G. Version A was the initial version where the controller and backend functions were implemented on a single minicomputer. Version A was implemented on a VAX-11/780 running the UNIX operating system. It included the request preparation and insert information generation functions of the controller and the directory management and record processing functions of the backends. The post-processing function was not implemented. Instead, the output was displayed directly from record processing. The disk input/output routines were omitted since they were operating-system-dependent, and subsequent versions of MDBS would have the PDP-11/44s (running RSX-11M) as the actual backends. Since the database was not to be stored on disks in this version, we implemented a pseudo-disk using the main memory. In addition, an interactive test interface was implemented.

Version B involved the development of a multi-process, multi-computer system with the same functionality as version A. The controller had three processes, request preparation, insert information generation, and post-processing. The backend had two processes, directory management and record processing. Concurrency control was added as a third process in a later version. Two computers were used, a VAX-11/780 (running VMS) for the controller, and a PDP-11/44 (running RSX-11M) for the backend.

Because of the multi-process, multi-computer structure, a message-passing facility was designed. Three types of message-passing facilities were defined: message passing within the controller, message passing within the backend, and message passing between the computers. The first two types are

categorized as intra-computer message passing, the last type as inter-computer message passing. The original explanation of the MDBS message passing facility can be found in [Boyn83]. The revised definitions of the MDBS messages is contained in Chapter 5 of this report.

As just described, version B used only a single backend. Thus we converted to two backends for version C. This version ran on three computers, a VAX-11/780 and two PDP-11/44s. However, it still lacked several required functions. There was no concurrency control. In addition, all the data including the database and its directory, were still stored in primary memories. Thus no disk input and disk output was required.

By changing from using a simulated disk in version C to an actual disk system, we obtained version D. This change, though logically simple, was difficult to implement, since it required the development of a low-level interface with the operating system of the PDP-11/44s. This interface was discussed in [Boyn83]. Version D included all the functions we had intended for our first real system, except concurrency control. Thus we next added a concurrency control process to give us version E. This process was also described in [Boyn83].

The next step in our implementation, version F, was to change directory management so that directory information is stored on the secondary memory rather than in the primary memory. This change is complex, since restructuring of the directory data is also required. The secondary-memory-based directory management of version F is described in [Boyn83].

The final version, version G, will incorporate access control in the backends and a friendly user-interface in the controller or host computer.

1.2. Concurrency Control Revisited

The main focus of this report is on the modification of the concurrency control process. Recall that directory data is used for the fast, efficient access of user data. In order to maintain both the consistency and integrity of the user data, we must also control access to directory data. Consequently, the concurrency control process developed in version E for controlling access to user data, must be expanded to include controlled access to directory data. In the rest of this report we describe in detail the implementation of version F, the multi-computer MDBS with concurrency control for

directory data. Chapter 2 contains an analysis of the concurrency control process for directory data. Chapter 3 describes the improvements made in the request execution of an update request. Chapter 4 describes the changes in the structure of directory management caused by the use of the secondary storage for directory data. Chapter 5 presents the revised definitions of the MDBS messages. Finally, Chapter 6 concludes this series of reports [Kerr82, He82, Boyn83] on the implementation of MDBS and presents a brief discussion of the next phase in the development of MDBS.

2. CONCURRENCY CONTROL IN DIRECTORY MANAGEMENT

In this chapter we discuss the concurrency control process for directory data. That is, we consider just how the access to attributes, descriptors, and cluster definitions must be controlled to preserve the consistency and integrity of the database. To motivate this discussion, some background information is presented.

MDBS is designed to perform the primary database operations, INSERT, DELETE, UPDATE, and RETRIEVE. Users access MDBS through the host by issuing either a request or a transaction. A request is a primary operation along with a qualification. A qualification is used to specify the information of the database that is to be accessed by the request. There are four types of requests, corresponding to the four primary database operations. An example of an update request would be:

UPDATE (FILE=Census and CITY=Cumberland) <POPULATION=40000>

which sets the population of Cumberland to 40,000. Notice that the qualification component of an update request consists of two parts, the query ((FILE=Census and CITY=Cumberland)) and the modifier (CITY=Cumberland). The query specifies which records of the database are to be updated. The modifier specifies how the records satisfying the query are to be updated [Hsia81a]. A user may wish to treat two or more requests as a transaction. In this situation, MDBS executes the requests of a transaction without permuting them, i.e., if T is a transaction containing the requests <R1><R2>, then MDBS executes the request R1 before request R2. Finally, we define the term traffic-unit to represent either a single request or a transaction in execution.

We recall that the directory information is stored in three tables: the attribute table (AT), the descriptor-to-descriptor-id table (DDIT) and the cluster-definition table (CDT). The attribute table maps directory attributes to the descriptors defined on them. A sample AT is depicted in Figure 3a. The descriptor-to-descriptor-id table maps each descriptor to a unique descriptor id. A sample DDIT is given in Figure 3b. The cluster-definition table maps descriptor-id sets to cluster ids. Each entry consists of the unique cluster id, the set of descriptor ids whose descriptors define the cluster, and the addresses of the records in the clusters. A sample CDT is shown in Figure 3c. Thus, to control access to directory data, we must control access to the AT,

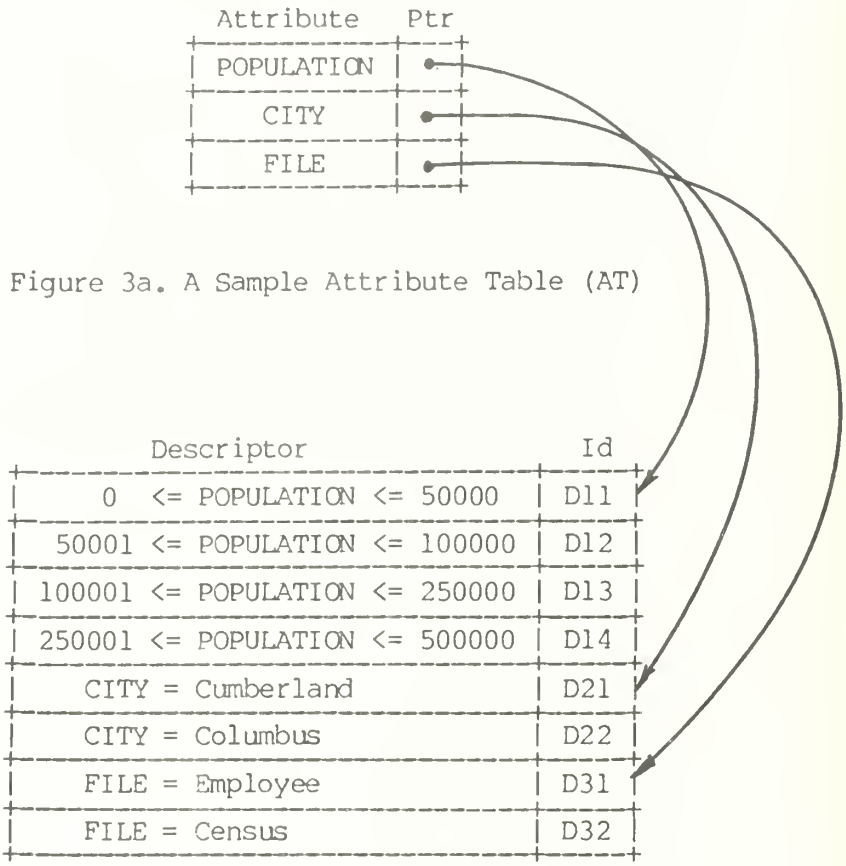


Figure 3a. A Sample Attribute Table (AT)

Dij = Descriptor j for attribute i.

Figure 3b. A Sample Descriptor-to-Descriptor-Id Table (DDIT)

Id	Desc-Id Set	Addr
C1	{D11,D21,D31}	A1,A2
C2	{D14,D22,D32}	A3

Figure 3c. A Sample Cluster-Definition Table (CDT)

DDIT, and CDT.

Lastly, we identify three classifications of descriptors. A type-A descriptor is a conjunction of a less-than-or-equal-to predicate and a greater-than-or-equal-to predicate, such that the same attribute appears in both predicates. An example of a type-A descriptor is as follows:

((POPULATION >= 10000) and (POPULATION <= 15000)).

A type-B descriptor consists of only an equality predicate. An example of a type-B descriptor is:

(POPULATION = 17000).

Finally, a type-C descriptor consists of the name of a type-C attribute. The type-C attribute defines a set of type-C sub-descriptors. Type-C sub-descriptors are equality predicates defined over all unique attribute values which exist in the database. For example, the type-C attribute CITY forms the type-C sub-descriptors

(CITY=Cumberland), (CITY=Columbus)

where "Cumberland" and "Columbus" are the only unique database values for CITY.

In the remainder of this chapter we first consider just why concurrency control for directory data is needed in MDBS. Then we examine the concurrency control process in the descriptor search phase. Lastly, we describe the concurrency control process in the cluster search phase.

2.1. The Need for Concurrency Control in Directory Management

To understand the need for controlling access to directory data, we review the execution sequence (without concurrency control) of a request (or a request of a transaction) when it is received by the backend. First, the directory management process determines the directory attributes for the request. This is the attribute search phase. Second, by looking up the directory attributes in the AT, directory management determines the descriptor id(s) for the request, i.e., the AT contains pointers to the DDIT, which contains the descriptor ids. This is the descriptor search phase. Using the descriptor id(s), directory management then determines the cluster id(s) of the cluster(s) that the request needs for execution. This is the cluster

search phase. The directory management process performs the address generation function and then sends the request to record processing for execution.

Access to user data is controlled by the database concurrency control process (DBCC), which was presented in [Boyn83]. When the DBCC receives a request from directory management, it attempts to lock all of the cluster(s) required by the request. Locking clusters involves a series of exercises in which access to certain entries of directory data tables is controlled. For example, if a cluster number in the CDT is locked, then other requests cannot access the Addr field of the CDT. Consequently, the numbered clusters cannot be accessed. This is done to insure the consistency of the user data. Before we examine why concurrency control is needed in the descriptor search phase, we observe that concurrency control is not needed for the attribute search phase. This occurs since attributes cannot be added to the database, rather they are defined when the database is loaded. However, new attribute values may be defined for an attribute if it is a type-C attribute.

Consider a user database consisting of three attributes, FILE, POPULATION and CITY, with the AT, DDIT, and CDT as in Figure 3a, 3b, and 3c, respectively. Note that FILE and CITY are type-C attributes, and that four type-A descriptors are defined for POPULATION. Suppose the name of Cumberland is to be changed to Slumberland through the request

UPDATE (FILE=Census and CITY=Cumberland) <CITY = Slumberland>.

Using the AT, directory management determines that the directory attributes for the request are FILE and CITY. Now the request enters into the descriptor search phase. Using the pointers from the AT, the descriptor search function determines that D32 is the descriptor id for (FILE=Census), and that D21 is the descriptor for (CITY=Cumberland). The insert generated by this update is

INSERT (<FILE,Census>,<POPULATION,58000>,<CITY,Slumberland>).

Since there is no descriptor for <CITY,Slumberland>, a new type-C subdescriptor id, D23, i.e., the id of descriptor 3 for attribute 2, is created for the pair <CITY,Slumberland>.

Now, suppose that a new request, RETRIEVE (CITY NOT= Boston) (CITY), arrives at the directory management process for processing. The predicate, (CITY NOT= Boston), specifies the restriction on which records are to be

retrieved. The clause, (CITY), specifies which attribute value is to be selected. Note that the new request needs all of the descriptors for the attribute CITY. Without concurrency control on the descriptor search phase of directory management, the following situation could arise. The retrieve request could find only D21 and D22 as descriptors for the attribute CITY. The update could then take place changing Cumberland to Slumberland causing the record to change to a new cluster, C3, defined by the descriptor id set {D14,D23,D32}. The retrieve request, however, will only retrieve those records of cluster C2, thus missing the newly updated record which also has the attribute value of the attribute CITY. Notice that the retrieve should not be allowed to do descriptor search until after the new descriptor id D23 had been created. In general, new type-C sub-descriptors may be generated for a type-C attribute, by an INSERT or an UPDATE request. Consequently, we must control access to the DDIT by locking the type-C attributes of the AT that appear in the request (see Section 2.2). If the type-C attributes of the AT are locked, the access of descriptor pointers by later request(s) is prohibited. Finally, let us examine why concurrency control is needed in the cluster search phase, by following the INSERT request defined above.

In the example above, recall that the descriptors defined for the INSERT are D32 for FILE, D12 for POPULATION and the newly created D23 for CITY. Notice that no cluster is defined in the CDT (Figure 3c) for the set of descriptors {D12,D23,D32}. Thus, a new cluster C3 is created for the set of descriptors {D12,D23,D32}. The address for C3 is assigned during the address generation phase. The record for the insert request, (Census,Slumberland,58000), is inserted into the secondary storage by record processing using the generated address.

Suppose that we are controlling access at the descriptor search phase. When the new request RETRIEVE (CITY NOT= Boston) (CITY) arrives at directory management, it must wait until the INSERT finishes descriptor search. When the INSERT request releases its lock on the directory attribute CITY, the new request locks CITY. Now, when the new request accesses the DDIT it will determine that D21, D22, and D23 are the required descriptors. But there is still a problem when the new request arrives at the cluster search phase. If cluster search for the new request occurs before C3 is created, then C1 and C2 are determined to be the required clusters. Once again, there will be an inconsistency in the data accessed for the RETRIEVE request. Therefore, we

must control access to the CDT by locking the descriptors of the DDIT. If a request cannot access a given descriptor, it cannot access the cluster ids associated with that descriptor.

Since uncontrolled access of the DDIT and CDT may lead to inconsistency, we have developed two concurrency control mechanisms. Descriptor search concurrency control (DSCC) controls access to the DDIT by locking directory attributes of the AT. Cluster search concurrency control (CSCC) controls access to the CDT by locking descriptors of the DDIT. Combining these two functions with DBCC yields what was labeled the concurrency control process in Figure 2. Lastly, Figure 4 is a pictorial description of how a request moves through the process structure of the backends.

2.2. Concurrency Control in the Descriptor Search (DSCC) Phase

In this section we examine the descriptor search concurrency control mechanism. We begin by considering the conditions under which the DDIT changes. The DDIT contains for the database three types of descriptors, type-A, type-B, and type-C. Recall that type-A and type-B descriptors, and type-C sub-descriptors are defined and stored in the DDIT at the database-load time. New type-A and type-B descriptors will not be created in the run-time environment. However, type-C sub-descriptors are generated and stored in the DDIT as new records with new values for type-C attributes are inserted in the database. So, we focus on the conditions under which new type-C sub-descriptors will be generated. Thus we examine the effect of type-C attributes appearing in the qualification component of a request.

2.2.1. Read and Write Control on Attributes

To control access to the DDIT, we lock the appropriate attributes of the AT. The retrieve and delete operations do not modify the DDIT. Retrieve and delete requests only read the information in the DDIT. Thus, for a retrieve or delete request, the type-C attributes needed by the request are locked for read access of the AT. In the previous section, we demonstrated that insert and update requests can modify the DDIT. If an insert request is inserting a type-C attribute value into the database, and no descriptor exists for that value then a new type-C sub-descriptor will be generated. Since there is no way to determine if a new descriptor will be generated for an insert request until the insert request tries to do descriptor search, the insert request

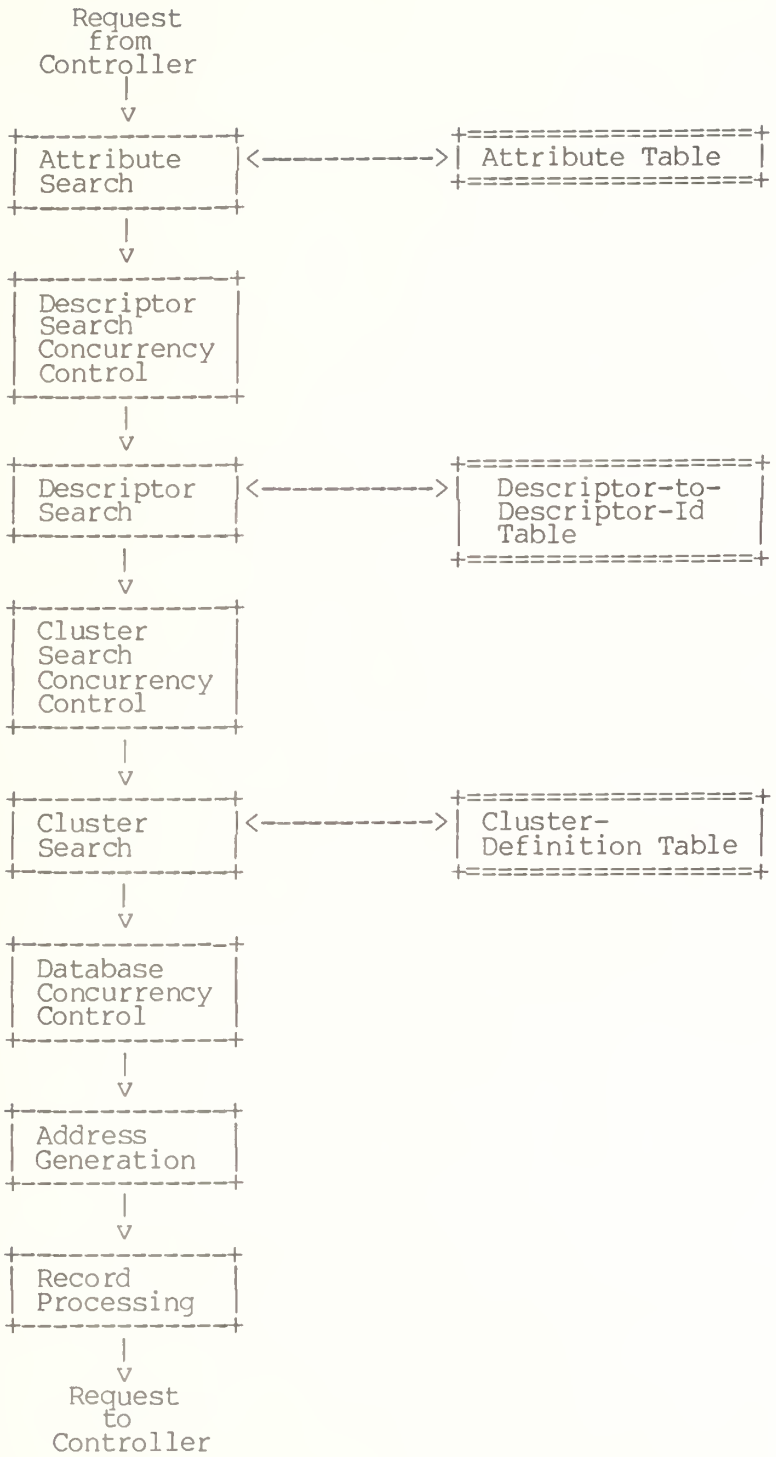


Figure 4. Three Levels of Concurrency Control in a Backend

must be granted write control on type-C attributes. So, the type-C attributes of an insert request are locked for write access of the AT.

In the previous section we also saw that the DDIT may be changed by an update request if the attribute listed in the modifier is a type-C attribute. In this situation, the update must be granted write control on the type-C attribute listed in the modifier, implying that the attribute is to be locked for write access of the AT. Additionally, an update request is granted read control on all type-C attributes listed in the query but not listed in the modifier. These attributes are locked as read access of the AT.

2.2.2. The DSCC Locking Scheme

The previous two paragraphs have described the standard read/write model for concurrent access to records of the database. The read/write model, often specified in database textbooks [Ullm82,Date83], can be characterized in three steps. First, multiple read locks on a record are permitted. Second, a write lock on a record excludes other reads and writes to that record. And, third, a write lock is granted on a record only if the record is not locked (for either read or write). In this context, a record is an entry of the directory table AT. With respect to the locking scheme of the AT, we conclude that type-C attributes of the AT can have either multiple read locks or a single write lock.

2.2.3. The DSCC Data Structures

We have developed two data structures to store the information needed for the descriptor search concurrency control mechanism. The traffic-unit-to-attribute table (TUAT), is a table internal to the descriptor search concurrency control mechanism (DSCC). The TUAT contains a list of traffic units and the type-C attributes needed by each request in each traffic unit. The mode of the request, either read or write, is also stored for each type-C attribute. This table is used to determine the status of any traffic unit. Additionally, this table keeps track of how many requests there are in a transaction. A sample TUAT table is shown in Figure 5a. The TUAT contains entries for four single requests and one transaction of two requests.

The second data structure, the attribute-to-traffic-unit table (ATUT) is used to keep track of which traffic unit(s) have requested locks on which type-C attribute(s). This table is essentially an inverse of the TUAT. This

Traffic-Units	Requests			
TU1 (one request)	A1 r	A2 r	A3 w	
TU2 (one request)	A3 w	A5 w		
TU3 (one request)	A2 r	A4 r		
TU4 (one request)	A1 r	A3 r	A4 w	
TU5 (two requests)	A2 r	A3 w	A4 r	

Ai = Attribute Name i
TUj = Traffic Unit j

MODE of Request

r = Read
w = Write

Note: Requests within a transaction are executed sequentially. The attributes of a request are separated from the attributes of another request by a bar.

Figure 5a. A Sample of the Traffic-Unit-To-Attribute Table (TUAT)

Type-C Attributes	Traffic-Units	
A1	TU1 r, TU4 r	TU1 and TU4 are performing descriptor search.
A2	TU1 r, TU3 r, TU5,R1 r	TU1, TU3 and TU5,R1 are performing descriptor search.
A3	TU1 w, TU2 w, TU4 r, TU5,R1 w	TU1 with exclusive write lock is performing descriptor search.
A4	TU3 r, TU4 w, TU5,R2 r	TU3 is performing descriptor search.
A5	TU2 w	TU2 with exclusive write lock is performing descriptor search.

TUm,Rn = Request n of Traffic Unit m

Figure 5b. The Attribute-To-Traffic-Unit Table (ATUT) corresponding to the TUAT in Figure 5a.

table contains a queue for each type-C attribute. Each attribute queue contains an entry for each of the requests requiring that attribute. Each entry contains an identifier for the request (the traffic unit and the request number) and the mode of access required (read or write). Figure 5b shows the ATUT corresponding to the TUAT shown in Figure 5a. At this point, we can now examine the descriptor search concurrency control mechanism.

2.2.4. The DSCC Lock Conversion Algorithm

When a traffic unit is ready for descriptor search, directory management sends a message to the descriptor search concurrency control mechanism. The message consists of a list of all type-C attributes needed by each request of the traffic unit, and the type of request, either INSERT, RETRIEVE, UPDATE, or DELETE. When such a message is received, DSCC stores the information for the traffic unit in the ATUT and TUAT, converting the request type to the corresponding mode, either read or write. Then the lock conversion process begins. For each attribute needed by each request of the new traffic unit, the lock conversion algorithm determines if the lock can be granted. If the lock is granted on an attribute, DSCC notifies directory management that the descriptor search on that attribute can begin. The process stops when the last attribute of the last request has been examined. Directory management notifies DSCC when the descriptor search on an attribute for a request is completed. For insert and update requests, all locks are released at once. For non-insert requests, the locks are released one at a time. DSCC then removes the attribute from the ATUT and the request from the TUAT, and attempts to grant locks for all other request(s) waiting for that attribute.

Now let's examine the lock conversion function. Suppose that a request R needs to lock an attribute A. The queue of the ATUT for attribute A is scanned. A pictorial description of the attribute-A queue is given below:

ATTRIBUTE A : Earlier Requests, R, Later Requests

There are two cases to consider; whether R needs a read lock or a write lock on attribute A. R can obtain a read lock on attribute A if

(a) R is the first request in the attribute-A queue,

or

(b) all earlier requests in the queue have locked attribute A for read access.

R can obtain a write lock on attribute A if and only if R is the first request

in the attribute-A queue. (Note: The special case of processing insert(s) generated by updates is examined in Chapter 3.) To fully understand the descriptor search concurrency control mechanism, we step through the algorithm using an example. Details of the algorithm are shown in Appendix B.

Suppose that the new traffic unit is TU5, which consists of two requests (see Figure 5b). The first request, R1, needs a read lock on A2 and a write lock on A3. The second request, R2, needs a read lock on A4. The lock conversion process first tries to determine if the read lock can be granted on A2, the first attribute needed by the first request of the traffic unit. Since the two earlier requests, TU1 and TU3, both have read locks on A2 (see A2 queue of ATUT, Figure 5b), the read lock on A2 for R1 of TU5 is granted, i.e., an attribute can have multiple read locks. DSCC notifies directory management that descriptor search can begin on the attribute A2. Now the algorithm tries to lock A3 for write access. Since R1 is not the first request in the ATUT queue for A3, the lock is not granted. Now the algorithm begins examining the second request. R2 will be granted a read lock on A4 only if all earlier requests in the A4 queue of the ATUT table have read locks. Since TU4 is requesting a write lock on A4 (see Figure 5b), the lock is not given to R2. Since all the attributes of each request have been examined, the algorithm stops.

2.3. Concurrency Control in the Cluster Search (CSCC) Phase

In this section we examine the cluster search concurrency control mechanism. We begin by considering the conditions under which the CDT changes. An entry of the CDT consists of the cluster number, the cluster definition, and the secondary storage addresses for the records in the cluster. The cluster definition is the set of descriptor ids whose descriptors define the cluster. Such a set is called a descriptor-id set. Descriptor-id sets are unique, and are used when referring to clusters. They are system data for permanent use. On the other hand, the descriptor search phase creates one or more descriptor-id groups for a request. A descriptor-id group is a collection of descriptor ids which define a set of clusters needed by the request. Thus descriptor-id groups are user data for one-time use. Since each cluster is defined by a descriptor-id set, we say that a descriptor-id group corresponds to the descriptor-id sets defined by the clusters needed by the request.

An insert request has exactly one descriptor-id group which corresponds to a unique cluster, i.e., a single descriptor-id set. A retrieve, delete, or update request can have multiple descriptor-id groups, and each group can correspond to multiple clusters (or descriptor-id sets). We denote descriptor-id sets by curly brackets ({...}), and descriptor-id groups by square brackets ([...]).

A new cluster is generated whenever there is a new record whose corresponding descriptor-id group is different from all the existing descriptor-id sets. Thus, to control access to the CDT, we lock descriptor-id groups. If a descriptor-id group is locked, then access to the cluster definitions is controlled. So, we need to determine what type of access the four primary database operations need on descriptor-id groups.

2.3.1. Read, Insert-Write, and Update-Write Control on Descriptor-Id Groups

The retrieve and delete operations do not modify the CDT. Retrieve and delete requests only read the information in the CDT. Thus, for a retrieve or delete request, the descriptor-id groups needed by the request are locked for read access. In an earlier section, we showed that an insert request can modify the CDT. If the insert request is inserting a record whose descriptor-id group does not correspond to an existing descriptor-id set, a new cluster will be created. We do not know if a new cluster will be created for an insert request until after cluster search, so, the insert request must be granted write access on its descriptor-id group. We refer to this as locking the descriptor-id group for insert-write control.

The last type of request, an update request, may also create a new cluster. In the previous section we presented an example of an update request that changed the attribute values in all records of the Census file with city equal to Cumberland to Slumberland. In this situation, a new type-C sub-descriptor, D24, for (CITY=Slumberland) was created. The descriptor-id group generated for the update request in the descriptor search concurrency control phase is [D2*,D32]. The descriptor D2* is used to represent all possible descriptors for the attribute being modified. Since there is no way to anticipate the creation of a new type-C sub-descriptor for the update request before the record processing phase, we represent all existing and possible future descriptors for the attribute city using D2*. Thus, [D2*,D32] represents a set of descriptor-id groups. Using this scheme we can logically

control access to any request that tries to use a cluster containing a descriptor for the attribute city. The descriptor-id group [D2*,D32] is a subset of the descriptor-id set {D11,D21,D32}, which defines cluster C2. The update request would need write access to the group [D2*,D32], which includes cluster C2, in order to prevent other requests from accessing cluster definitions associated with that group until the update request is completed. This prevents other requests from accessing cluster definitions which are supersets of [D2*,D32]. Thus, an update request must be granted write control on its descriptor-id group(s). We refer to this as locking the descriptor-id group(s) for update-write control.

2.3.2. The CSCC Locking Scheme - The Notion of Conflict-Free

The differentiation between the insert-write and update-write locks is mandated by the complexity of the cluster search locking algorithm. Instead of comparing single units, we compare descriptor-id groups. We begin with a definition. Two descriptor-id groups are said to be conflict-free if

(a) both descriptor-id groups require read locks,

or

(b) one or both descriptor-id groups require write locks and they do not define the same cluster.

Now let us discuss how to determine if two descriptor-id groups are conflict-free. There are two cases to consider depending on whether or not one of the requests is an insert.

Two descriptor-id groups for non-insert requests are conflict-free if they contain different descriptors for a common attribute. This occurs since a cluster cannot contain two descriptors for an attribute, i.e., it is therefore not possible for the two descriptor-id groups to be subsets of the same cluster. As an example, the two descriptor-id groups [D11,D22] and [D11,D23] are conflict-free since they have different descriptors for attribute 2, i.e., D22 and D23. Conversely, the two descriptor-id groups [D11,D22] and [D11] are in conflict since [D11] is contained in [D11,D22] and therefore the groups can be in the same cluster. Further, observe that [D11] and [D22] are also in conflict, since there may be a cluster containing them both.

If one or both of the descriptor-id groups represents an insert request, the test for conflict-free is different since the descriptor-id group for an insert request represents a unique cluster. If both requests are inserts,

then the descriptor-id groups are conflict-free if the descriptor-id groups are not identical. If one of the requests is a non-insert request, then the descriptor-id groups are conflict-free if the descriptor-id group for the non-insert request is not contained in the descriptor-id group for the insert request.

2.3.3. Two Categories of Locks

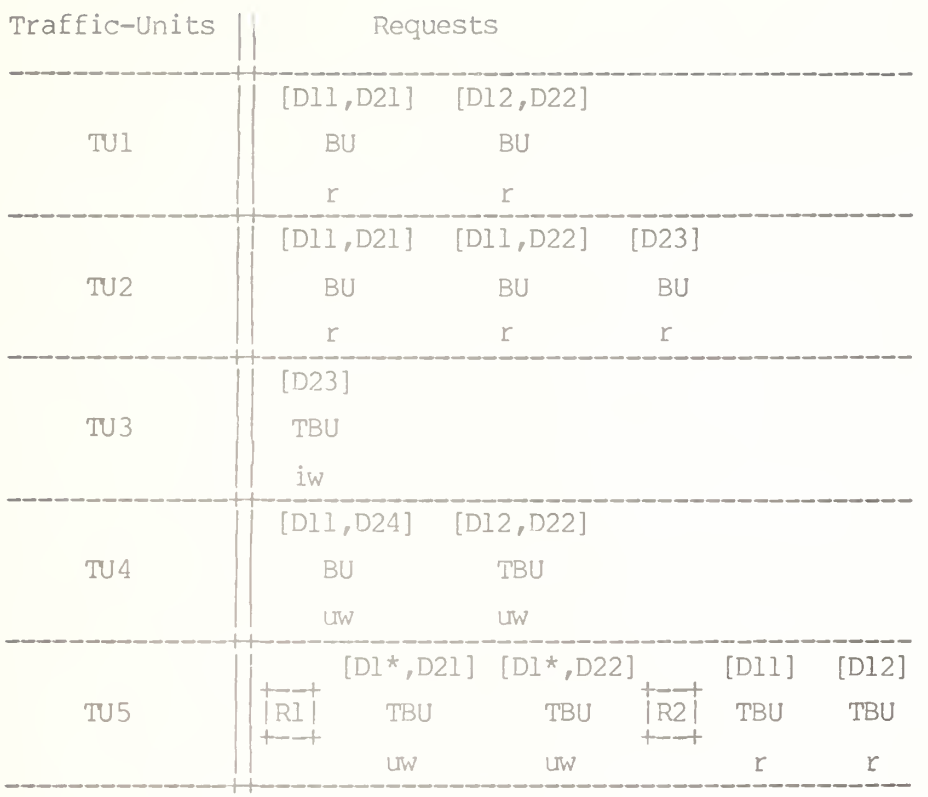
To keep track of which descriptor-id groups have obtained either a read, insert-write, or update-write lock, we introduce two categories of locks on descriptor-id groups: "to-be-used" and "being-used". As soon as a request reaches a backend, it locks the descriptor-id group(s) it needs in the "to-be-used" category. The "to-be-used" category of locks secures the request's claim for a "being-used" lock on a descriptor-id group. In this way, we can prevent a later request from locking a descriptor-id group for which an earlier request is waiting. Before the request can do cluster search, the locks on all descriptor-id group(s) must be converted to the "being-used" category. The "being-used" lock signifies that a request has access to a descriptor-id group. A "being-used" lock is granted on a descriptor-id group if that group is conflict-free with all earlier descriptor-id group(s).

2.3.4. The CSCC Data Structure

To store the information needed by the cluster search concurrency control mechanism, the traffic-unit-to-descriptor-id-groups table (TUDIGT) was developed. The TUDIGT contains a list of traffic units and the descriptor-id groups needed by each request in each traffic unit. The mode, either read, insert-write, or update-write, and the category, either "to-be-used" or "being-used", of each descriptor-id group is also stored. Figure 6 shows a sample TUDIGT which contains entries for four requests and one transaction of two requests. We now examine the cluster search concurrency control mechanism.

2.3.5. The CSCC Lock Conversion Algorithm

When a traffic unit is ready for cluster search, directory management sends a message to the cluster search concurrency control mechanism. The message consists of a list of all descriptor-id groups needed by each request of the traffic unit, and the type of request, either INSERT, RETRIEVE, UPDATE, or DELETE. The information for the new traffic unit is stored in the TUDIGT, with the request type converted to the appropriate mode, i.e., either read,



TUi = Traffic Unit i
Rj = Request j

MODE of Descriptor-Id Group

r = Read
iw = Insert-Write
uw = Update-Write

CATEGORY of Request

TBU = To-Be-Used
BU = Being-Used

Note : Traffic units TU1 and TU2 are currently performing cluster search. TU3 must wait until TU2 finishes, so that it can lock [D23]. The first group of TU4, [D11,D24] is conflict-free with all earlier groups. However, the second group, [D12,D22], conflicts with the TU1. TU5, which contains two requests, the first of which is an update, also is waiting.

Figure 6. A Sample Traffic-Unit-To-Descriptor-Id-Groups Table (TUDIGT)

insert-write, or update-write. After the information for the new traffic unit has been stored, the lock conversion process begins. For each descriptor-id group needed by each request of the new traffic unit, the lock conversion algorithm determines if the lock can be granted. A lock on a descriptor-id group can be granted if that group is conflict-free with all earlier descriptor-id groups. If all locks for all descriptor-id groups for a given request are converted to "being-used", CSCC notifies directory management to begin cluster search. The process stops when the last descriptor-id group of the last request has been examined. Directory management notifies CSCC when the cluster search for a request has completed. CSCC removes the information for the request from the TUDIGT and attempts to grant locks for all other waiting request(s).

Once again, we step through the algorithm using an example. Suppose that the new traffic unit is TU4, which consists of one request (see Figure 6). We will assume that the information for TU5 has not been received by CSCC. The first descriptor-id group [D11,D24], needs an update-write lock. We compare [D11,D24] with all earlier descriptor-id groups. [D11,D24] is conflict-free with [D12,D22] since D11 and D12, descriptors for attribute 1, are different. [D11,D24] is conflict-free with [D11,D21], [D11,D22] and [D23] since the descriptor for attribute 2, D24, is different from D21, D22, and D23. Thus, the "being-used" lock is granted since [D11,D24] is conflict-free with all earlier requests. Now the algorithm tries to obtain an update-write lock on the second descriptor-id group, [D12,D22]. While [D12,D22] is conflict-free with [D11,D21] of TU1, [D12,D22] conflicts with [D12,D22] of TU1. Therefore, the lock is not granted. Since all descriptor-id groups for the request have been examined, the algorithm stops.

There are two final notes on the cluster search concurrency control mechanism. First, the detailed design of CSCC can be found in Appendix B. Second, the special case required to handle insert(s) caused by an update request is examined in Chapter 3.

3. THE REQUEST EXECUTION OF AN UPDATE REQUEST

In this chapter we examine the execution sequence of an update request. An update request modifies records of the user database. Under normal circumstances, an update request will retrieve a record from the user database, update the specified record value, and write the record back to the secondary storage. The update request will continue until all appropriate records have been modified. However, under some conditions the update of a record is logically equivalent to the retrieval of the record, the deletion of the record, the creation of a new record, and the insertion of the new record. In such a situation, an insert request must be generated. Thus, processing an update request must proceed in two logical phases. First, all the records to be updated are retrieved and either modified or converted to insert requests. This is the record-modification phase of the update request. Second, the insert requests are performed. This is the generated-insert phase of the update request. To begin, we focus on the conditions under which an update request generates an insert request.

Suppose that a user wants to increase all populations between 10,000 and 40,000 people by 15,000 people in the Census file. The update for this request is listed below:

```
UPDATE ((FILE = Census) and
        (POPULATION >= 10000 and POPULATION <= 40000))
        <POPULATION = POPULATION + 15000>
```

In referring to Figure 3b, the descriptors needed by the request are D31 for the clause (FILE = Census) and D11 for the clause (POPULATION >= 10000 and POPULATION <= 40000). The cluster corresponding to these descriptors would be C1, which is defined by the descriptor-id set {D11,D21,D31}. There are two cases to consider. First, suppose that there is a record in the user database with population 12,000. The record will be modified by the update request, changing the record value for population to 27,000 (i.e., 12,000 + 15,000). The descriptor id for this record value is still D11, so the modified record is written back to the secondary storage.

Now, assume that there is a record in the user database with population 37,000. This record will be modified by the update request, changing the record value for population to 52,000 (i.e., 37,000 + 15,000). The descriptor id for this record is now D12, 50001 <= POPULATION <= 100000. But notice that

there is not a cluster defined for this record (see Figure 3c). Thus a new cluster id corresponding to the descriptor ids, D12, D21 and D31, would be entered into the CDT. When a record moves from one cluster to another cluster (either an existing cluster or a new cluster), as a result of the update action, we say that the record has changed cluster. In this example, the record must change into a new cluster which results in the definition of a new cluster. However, the fact that a new cluster is created is not important. The key point is that the record changed cluster. Since the record has changed cluster, it cannot simply be written back to the secondary storage with the modified record value. Instead, the old record will be marked for deletion and an insert request for the new record will be generated. (Note: In this example a new cluster is defined from existing descriptor ids.)

In general, an update request will generate an insert request if the record being modified changed cluster. The insert request generated as a result of the update request is characterized as a generated-insert request. In the example above, the generated-insert request would be:

INSERT (<FILE = Census>,<POPULATION = 52000>,<CITY = Cumberland>)

The record to be inserted contains the modified value of the population attribute along with the values of city and file stored in this record. The record values, Census for file (descriptor D32), 52,000 for population (descriptor D12) and Cumberland for city (descriptor D21) define the cluster {D12,D21,D32} for this record. Thus, an update request can consist of the two logical phases specified above. The remainder of this chapter examines the two logical phases, and how generated-insert requests are processed by the descriptor search, cluster search and database concurrency control mechanisms.

3.1. The Two Phases of an Update Request

In this section we examine the two phases of an update request, the record-modification phase and the generated-insert phase. It would be desirable to overlap these two phases. Otherwise the insert requests must be stored for later processing. Records may be inserted into a cluster as soon as the record-modification phase for that cluster has completed. The rest of this section is divided into two parts. First we examine the execution sequence for an update request and its generated-insert requests. This analysis does not assume any overlap between the record-modification and generated-insert

phases. Second, we examine how the overlap of the two phases can be implemented.

To motivate this discussion, we recap the execution sequence of a traffic unit (consisting of one or more requests). The traffic unit is received by the controller from the host. After processing the traffic unit in the controller, request preparation sends the traffic unit to the directory management process of each backend for execution. Each request in the traffic unit moves through the backend (see Figure 4) finally reaching the record processing process. Record processing manages the physical data operations, i.e., inserting a new record for an insert request, retrieving records for a retrieve request, etc. When a request has finished processing, record processing sends the results to the post processing function of the controller. Post processing collects the results for a traffic unit and forwards them to the host.

3.1.1. Execution of an Update Request without Overlap

We begin by examining how an update request is processed by record processing. We are assuming that there is no overlap of the record-modification and generated-insert phases. After the database concurrency control mechanism determines an update request can execute, directory management generates the cluster addresses needed by the update request. Record processing cycles through the clusters track by track, examining all records which satisfy the update request, i.e., the records which satisfy the query component of the update request. After retrieving a record, and determining that the record satisfies the query, record processing recalculates the specified record value using the modifier. Then, record processing sends the attribute being modified, along with the old and new record values for this attribute, to directory management to determine if the record has changed cluster.

There are two cases to consider. If the updated record has not changed cluster, the modified record is written back to the secondary storage. If the modified record changed cluster, the old record is marked for deletion, and the modified record is sent by record processing to request preparation (of the controller), i.e., the "changed-cluster-record" message in Figure 7, so that an insert request for that record can be generated. Now we follow the actions taken by a generated-insert request.

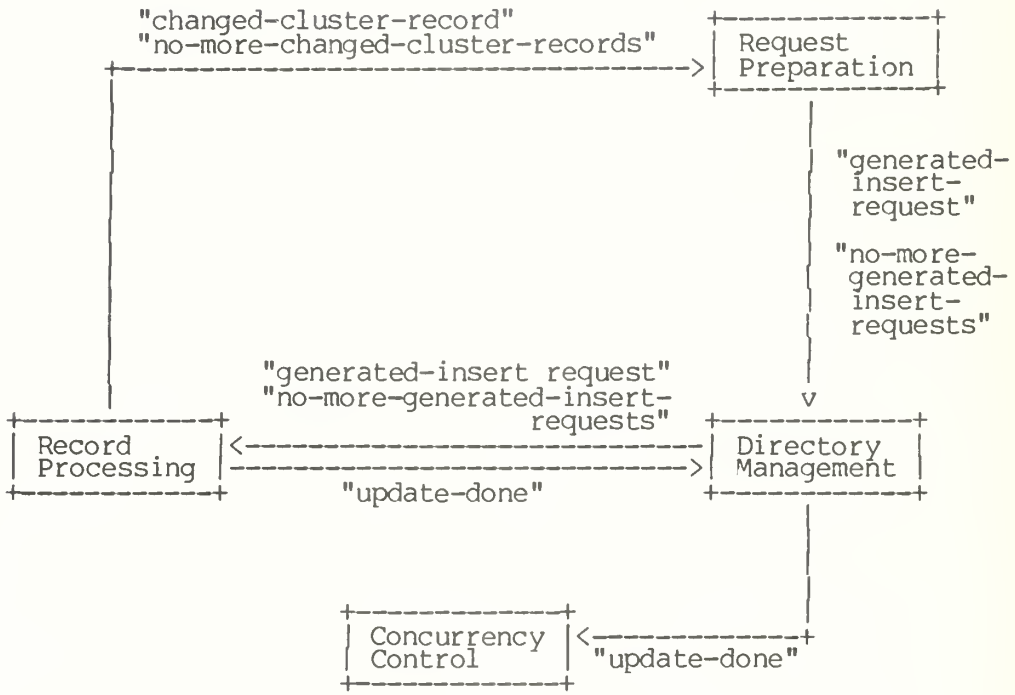


Figure 7. Messages for the Request Execution of an Update Request

After request preparation receives the "changed-cluster-record" message (see Figure 7) from record processing, it generates an insert request. The generated-insert request is broadcast to the directory management process of every backend (the "generated-insert-request" message from request preparation to directory management in Figure 7). This request is the same as an insert request, except that it is marked so that it may be associated with the update request that caused it. The generated-insert request flows through the process structure of the backend (see Figure 4). However, since this insert is associated with a unique update request, the actions of the generated-insert request in the descriptor search, cluster search, and database concurrency control mechanisms must be processed as a special case.

The last stages of the record-modification phase in processing an update request occur when all records satisfying the query have been examined. The record processing process sends a message informing request preparation that there will be "no-more-changed-cluster-records" generated (see Figure 7). Request preparation keeps track of the generated-insert requests since an insert request may be generated by any backend, and an insert request generated at one backend may be carried out at another backend. When request preparation has received the "no-more-changed-cluster-records" message from every backend, the record-modification phase of the update request has finished. Request preparation notifies the directory management process of every backend that there will be "no-more-generated-insert-requests" (see Figure 7). When the directory management process of a backend has sent all generated-insert requests to record processing, and has also received the "no-more-generated-insert-requests" message from request preparation, directory management sends a "no-more-generated-insert-requests" message (see Figure 7) to record processing. Finally, when record processing has finished executing all of the generated-insert requests and has received the "no-more-generated-insert-requests" message from directory management, it sends an "update-done" message (see Figure 7) to directory management. Directory management releases space being used by the update request, and then notifies concurrency control (with the "update-done" message) that it can release locks being held by the update request.

3.1.2. Execution of an Update Request with Overlap

When deciding how to process a generated-insert request, we were faced with the problem of when to store the new records created by the generated-insert requests. These records must wait until the record-modification phase has finished. If the record for a generated-insert request is allowed to be placed in the secondary storage by record processing before the record-modification phase has finished, the record-modification phase may modify the newly inserted record. Such a modification would result in an inconsistent user database. We have two places where the records created by the generated-insert requests can be temporarily stored, in either directory management or record processing. In chosing to store them in record processing, we permit the records to progress through the system as far as possible.

We also discovered that we could easily increase the "intelligence" of the record processing process. Recall that when processing an update request, record processing examines the records track by track. When record processing has finished examining a track, insertion of records of generated-insert requests into that track can begin. Such a generated-insert request will not have to wait until the record-modification phase of the update request is done before placing new records on the secondary storage. Thus the retrieval and generated-insert phases can be successfully overlapped.

In the current version of MDBS we have yet to implement procedures to handle the overlap of the two phases. However, we have developed our design so that the eventual processing of the record-modification and generated-insert phases concurrently can be implemented without any major modifications. Lastly, we examine the steps required to process generated-insert requests.

When database concurrency control determines that an insert (any insert) request can execute, directory management generates an address for the request. Directory management then forwards the insert request and address to record processing. Record processing checks to see if the insert request is a generated-insert request. If it is, record processing will hold the insert request until the record-modification phase of the associated update request has completed. Otherwise, record processing executes the insert request.

3.2. Concurrency Control for Generated-Insert Requests

In this section we examine the steps taken to process a generated-insert request in the descriptor search, cluster search, and database concurrency control mechanisms. This section is divided into two parts. First, we consider the various design issues when developing a strategy to process generated-insert requests. These issues are reviewed for the three concurrency control mechanisms. Second, we examine the implementation details for processing generated-insert requests in the three concurrency control mechanisms.

3.2.1. The Design Issues

We begin this section by presenting the general idea involved in the processing of a generated-insert request. A generated-insert request is caused by a particular update request. The update request has locks on all attributes, descriptor-id groups, and clusters needed by the generated-insert request. When the generated-insert requests of an update request are attempting to secure locks (on attributes, descriptor-id groups, or clusters), they have a logical priority over other later requests trying to obtain the same locks. This is due to the fact that the generated-insert requests represent the second phase in the execution of an update request, i.e., the generated-insert requests are part of the update request.

However, we still must be careful when processing generated-insert requests of the update request. In the descriptor search concurrency control mechanism, generated-insert requests cannot execute concurrently since they may conflict with each other, e.g., two generated-insert requests may both cause the generation of the same new descriptor id. In the cluster search concurrency control mechanisms, generated-insert requests are also prohibited from concurrent execution, since they may create new clusters. However, in the database concurrency control mechanism, generated-insert requests of an update request are allowed to execute concurrently. This occurs since two or more generated-insert requests are always compatible, i.e., the consistency of the user database will not be affected if the generated-insert requests are executed concurrently. The remainder of this section analyzes the design issues for processing generated-insert requests in the three concurrency control mechanisms, using an example which is developed in the next subsection.

(A) On Descriptor Search Concurrency Control

Suppose that we are processing the update presented in Section 2.1,

UPDATE ((FILE=Census) AND (CITY=Cumberland)) <CITY=Slumberland>.

This update changes the attribute values in all records of the Census file with city Cumberland to Slumberland. When the update request is processed, the descriptor search concurrency control mechanism will lock the attributes FILE and CITY of the ATUT. FILE is locked for read access and CITY is locked for write access by the update request. When the update request finally reaches record processing, records for the appropriate cluster(s) are retrieved. When a record containing the values (Census,Cumberland) is found, the CITY record value is changed to Slumberland. Since this record changed cluster, a generated-insert request is created, say,

INSERT_1 (<FILE,Census>,<CITY,Slumberland>,<POPULATION,record_value_1>),

where record_value_1 is the record value for POPULATION. To fully illustrate the problem, let us suppose that a second generated-insert request was created, say,

INSERT_2 (<FILE,Census>,<CITY,Slumberland>,<POPULATION,record_value_2>),

where record_value_2 is another record value for POPULATION. Both of these requests will have been sent to descriptor search concurrency control.

The generated-insert request, INSERT_1, will be processed first in descriptor search concurrency control. Assume that INSERT_1 will be granted locks on the attributes FILE, CITY, and POPULATION. DSCC will notify directory management that it may do descriptor search for INSERT_1. At this point, since there is no descriptor id for (CITY=Slumberland), a new type-C subdescriptor, with id D23, will be defined. Notice that INSERT_2 also needs access to this new descriptor, so it must wait until INSERT_1 has finished descriptor search before it can lock CITY. Thus, these two generated-insert requests cannot be executed concurrently.

Lastly, it is important to examine the types of locks given to INSERT_1 on the attributes FILE, CITY and POPULATION. In Section 2.2.1 we concluded that an insert request needs write access on all attributes needed by the request. However, a generated-insert request is a special type of insert

request. The update request is holding locks on the attributes needed by the generated-insert request. One of the attributes, CITY (the attribute being modified), is being held as write access. FILE is being held by the update request for read access. Since there is no way that the generated-insert request INSERT_1 can change the value of FILE, i.e., create a new descriptor, it is only necessary that this insert request has read access on FILE. Additionally, we know that the update request does not hold a lock on POPULATION. Once again there is no way that the generated-insert request can create a new value for POPULATION. In this situation, since the update is not locking POPULATION, we ignore the attribute, and simply notify directory management that descriptor search for the request can commence. Now let us see whether or not there is any problem in the cluster search concurrency control mechanism.

(B) On Cluster Search Concurrency Control

We first assume that the record values for POPULATION, record_value_1 and record_value_2, are represented by the same descriptor, say D11. Working with the two generated-insert requests INSERT_1 and INSERT_2, we find that INSERT_1 and INSERT_2 will have the descriptor-id group, [D11,D23,D32], after the descriptor search phase. Recall that for insert requests, a descriptor-id group defines a unique cluster. Since their descriptor-id groups are identical, both requests, INSERT_1 and INSERT_2, will define the same cluster. Since the cluster {D11,D23,D32} does not yet exist, only one request, INSERT_1, can be passed to directory management for cluster search. INSERT_2 must wait since a new cluster is to be defined. Once again, we cannot allow concurrent execution of generated-insert requests.

(C) On Database Concurrency Control

Finally, we arrive at the database concurrency control mechanism. At this point, we assume that a new cluster, C5, corresponding to {D11,D23,D32} has been created. The first thing we observe is that the update request is only locking the cluster C1, since C5 did not exist at the time the update request locked C1. When the first generated-insert request arrives, it asks for C5. We first lock C5 for the update request. Both INSERT_1 and INSERT_2 are requesting locks on C5. Since insert requests are compatible, the locks

for both requests are also granted. The requests can now be executed concurrently, since the records for INSERT_1 and INSERT_2 will be inserted into different blocks of the same or different track.

(D) Conclusions on Generated-Insert Requests

We must take special care when processing generated-insert requests. We have a special two-step procedure when handling generated-insert requests. First, we must include the generated-insert request as part of the original traffic unit, i.e., the traffic unit which contains the update request that caused the generated-insert request. Such a step is necessary since the generated-insert request is the second phase of the update request. Second, depending upon the concurrency control mechanism, we have different degrees of concurrency for the generated-insert requests. In the descriptor search concurrency control mechanism only one request may write to the DDIT. In the cluster search concurrency control mechanism, only one generated-insert request may write to the CDT. In the database concurrency control mechanism, we permit multiple writes to the user database.

3.2.2. The Implementation Details

In this section we investigate the two-step process of handling generated-insert requests in the DSCC, CSCC, and DBCC respectively. The two steps are, one, entering the information for a generated-insert request into the concurrency control data structures, and two, assigning locks for a generated-insert request.

(A) Processing Generated-Insert Requests in DSCC

In this section we examine the steps taken to process a generated-insert request in the descriptor search concurrency control mechanism. There are two main differences when processing a generated-insert request. First, the information for the generated-insert request must be entered into the TUAT and ATUT data structures in the designated place. Second, the locking scheme for a generated-insert request varies slightly from the one described in Section 2.2.2. We begin by considering how the information for the generated-insert request is entered into the TUAT and ATUT.

The generated-insert request is received by the descriptor search concurrency control mechanism with a list of the type-C attributes needed by the request and a code associating the request with a unique traffic unit and the update request within the traffic unit that caused the generated-insert request. The list of type-C attributes needed by the first generated-insert request (of a particular update request) is entered into the TUAT after the corresponding update request and before any later requests of the traffic unit. Entries for subsequent generated-insert requests are inserted after earlier generated-insert requests and before any later requests of the traffic unit.

In general, an insert request must lock all its attributes for write access. However this is not the case for a generated-insert request. As an example, suppose that the generated-insert request was caused by the first request in traffic unit TU3. The portion of the TUAT table showing TU3 is reproduced below.

Traffic-Units	Requests			
TU3	A2	A3	A4	
	w	r	r	

Further suppose that the generated-insert request needs write access on attributes A2, A3, and A4. However, since the update request is locking attribute A3 for read access, attribute A3 for the generated-insert request only requires read access, i.e., there is no way that a new descriptor can be created on attribute A3. Thus descriptor search concurrency control will change the write access to read access before entering A3 into the TUAT. When the information for the generated-insert request is inserted into the TUAT, the TU3 queue now appears as:

Traffic-Units	Requests						
TU3	A2	A3	A2	A3	A4	A4	
	w	r	w	r	w	r	

The generated-insert request must be placed after the update request (and any other earlier generated-insert requests for this update request) and before any later requests, since the generated-insert request is part of the update request, and must be processed after any earlier generated-insert requests and before any later requests. This is done to insure the consistency of the

directory information, specifically the DDIT data structure.

The information for the generated-insert request must also be entered into the ATUT. For each type-C attribute needed by the generated-insert request, an entry consisting of the traffic unit number, the request number, and the mode of access (either read or write), is created. Each of these entries is inserted into the corresponding attribute queue of the ATUT in the following manner. If the update request request also needed this attribute, the entry is inserted into the ATUT after the entry for the update request (and any other entries for earlier generated-insert requests for this update request) and before entries for any later requests of the same or different traffic unit. If the update request did not need this attribute, the entry is discarded.

Following through with the example above, the portion of the ATUT before the entries for the generated-insert request are added is:

Type-C Attributes	Traffic-Units
A2	TU3,R1 w
A3	TU2 TU3,R1 r r
A4	TU1 TU2 TU3,R2 r w r

After the entries for the generated-insert are added, the ATUT table is

Type-C Attributes	Traffic-Units
A2	TU3,R1 TU3,RG w w
A3	TU2 TU3,R1 TU3,RG r r r
A4	TU1 TU2 TU3,R2 r w r

where RG denotes the generated-insert request. The RG entry for attribute A4 was discarded, since the original update request didn't lock A4. After entering the generated-insert request into the TUAT and ATUT, the processing of the request begins. The processing of a generated-insert request is the same as any other request (see Section 2.2.4) except for the lock conversion function.

Suppose that the generated-insert request RG needs to lock attribute A. The queue of the ATUT for attribute A is scanned (shown below).

ATTRIBUTE A : Earlier Requests, RG, Later Requests

There are two cases to consider:

(a) when the earlier request adjacent to RG is the update request that caused it,

or

(b) when the earlier request adjacent to RG is another generated-insert request.

In case (a) we have two possibilities, whether RG is requesting a read or write lock on attribute A. For either possibility, since RG is the first insert generated by the update request, the lock, either read or write, is granted, regardless of the type of lock being held by the update request. In case (b), the lock is not granted since a new descriptor may be created by the earlier generated-insert request currently holding the lock. The net effect of this locking scheme is the serialization of the generated-insert requests of an update request. In the example given above, the generated-insert request will be granted a write lock on A2 and a read lock on A3.

At first glance, there seems to be a serious problem with case (a). When the update request adjacent to the generated-insert request has a write lock, we are also granting the generated-insert request a write lock. This seems to contradict the standard read/write model described in Section 2.2.2. There are two key observations to make here. First, we are only using the update request to retain the lock on a specific attribute. Second, for a generated-insert request to arrive at DSCC, the update request must be currently in record processing, i.e., finished with descriptor search. The generated-insert request is logically part of the update request, and must be allowed to perform descriptor search before any later requests. Also, since the update request has finished descriptor search, there is no chance that inconsistency may develop in the DDIT. Therefore, we use the update request to hold the lock for any of its generated-insert requests.

(B) Processing Generated-Insert Requests in CSCC

This section examines the steps taken to process a generated-insert request in the cluster search concurrency control mechanism. Once again,

there are two main differences when processing a generated-insert request; entering the generated-insert request into the TUDIGT (see Figure 6) and processing the generated-insert request through the lock conversion scheme.

The generated-insert request is sent to cluster search concurrency control with the descriptor-id group needed by the request, the mode of the request (insert-write), and a code associating the request with a unique traffic unit and the update request within that traffic unit that caused the generated-insert request. The descriptor-id group for the first generated-insert request (of a particular update request) is entered into the TUDIGT after the corresponding update request and before any later requests in the traffic unit. Descriptor-id groups for subsequent generated-insert requests are inserted after earlier generated-insert requests for that update request and before any later requests in the traffic unit. The situation here is the same as described in the previous section for the TUAT table. After entering the generated-insert request into the TUDIGT, the processing of the request begins. The processing of a generated-insert request is the same as any other request (see Section 2.3.5) except for the lock conversion scheme.

The locking scheme is somewhat simplified for a generated-insert request. As described in the previous section, we are just using the update request to hold the lock on its descriptor-id group for any generated-insert requests. Also, recall that an update locks its descriptor-id group(s) for update-write access (see Section 2.3.1). The generated-insert request will be granted an insert-write lock on its descriptor-id group only if it is adjacent to the update request that caused it. Other later generated-insert requests will not be granted an insert-write lock, since their descriptor-id groups will conflict with the generated-insert request currently holding an insert-write lock. In fact, the descriptor-id group for the generated-insert request holding the insert-write lock will conflict with any other generated-insert request's descriptor-id group on the attribute being modified, i.e., the two groups will not be conflict-free. Once again, we are guaranteeing the serialization of the generated-insert requests.

(C) Processing Generated-Insert Requests in Database Concurrency Control

This section examines the steps taken to process a generated-insert request in the database concurrency control mechanism. Once more, there are

two main differences when processing a generated-insert request; entering the information into the traffic-unit-to-cluster table (TUCT) and cluster-to-traffic-unit table (CTUT) (see [Boyn83] for a description of these data structures and an explanation of the database concurrency control algorithm), and processing the generated-insert request through the lock conversion function.

The generated-insert request is sent to database concurrency control with a cluster needed by the request, and a code associating the request with a unique traffic unit and the update request within that traffic unit that caused the generated-insert request. An entry of the TUCT for a generated-insert request consists of the cluster needed by the request, the type of the request (an insert), and the category of lock being held ("to-be-used"). The entry for the first generated-insert request (of a particular update request) is inserted into the TUCT after the corresponding update request and before any later requests in the traffic unit. Entries for subsequent generated-insert requests are entered into the TUCT after earlier generated-insert requests for that update request and before any later requests in the traffic unit.

The information for the generated-insert request is also entered into the CTUT. For the cluster needed by the generated-insert request, an entry consisting of the traffic unit number, the request number, the type of request (insert), and the category of lock being held ("to-be-used"), is inserted into the corresponding cluster queue of the CTUT. If the update request which caused the generated-insert request also needed this cluster, this entry is inserted into the CTUT data structure after the entry for the update request that caused it (and any other earlier generated-insert requests for this update request) and before entries for any later requests of the same or different traffic unit. If the update request did not need this cluster, i.e., a new cluster was defined, then an entry for the update request is created (locking the cluster as "being-used"), and entered into the cluster queue. The entries for generated-insert requests are entered at the end of this cluster queue. The processing of a generated-insert request in the database concurrency control mechanism is the same as any other request (see [Boyn83]) except for the lock conversion scheme.

Briefly, the locking scheme for the database concurrency control mechanism tries to convert locks on clusters needed by a request from "to-be-used"

to "being-used". If a "being-used" lock is not granted, a "waiting" lock is assigned to the request for that cluster. The "waiting" lock secures the request's claim for a "being-used" lock on a cluster. If all locks on clusters needed by a request are converted to "being-used", the request is passed to directory management. Directory management does the address generation for the request and forwards the request and generated address(es) to record processing. Generated-insert requests are compatible. Since the update request has secured the lock on a cluster, a generated-insert request is given a "being-used" lock on the cluster that it needs. Thus, the locking scheme is very straight-forward.

4. THE SECONDARY-MEMORY-BASED DIRECTORY MANAGEMENT

In this chapter we describe the implementation of directory management using the secondary storage. Let us first recall the main functions of directory management. Directory management receives traffic units from the controller. Directory management processes the traffic unit one request at a time. Each request passes through a number of phases under the control of the directory management process. These phases are: attribute search, descriptor search, cluster search, and address generation (see Figure 4 again). To proceed through these phases, directory management accesses the directory data, i.e., the attribute table (AT), the descriptor-to-descriptor-id table (DDIT), and the cluster-definition table (CDT).

Version A through version E stored the directory data in the primary memory. In the final version, version F, the directory data is stored in the secondary storage. When the directory data is in the secondary storage, processing is more complex because there is a delay every time some directory data is to be read from or written to the secondary storage. Additionally, when a new type-C sub-descriptor is created, a new cluster is defined, or an address of a new record is allocated, the insertion of new directory data into the tables maintained on the secondary storage must be performed. Thus, in the following sections we describe the processing required for each phase of the secondary-memory-based directory management, attribute search, descriptor search, cluster search and address generation. Appendix D contains an analysis of the algorithms required for the insertion of new directory data. To simplify the discussion, we introduce some new notation and concepts.

In the attribute search phase, we process the query component of a request one predicate at a time. For each predicate, we must determine the attribute-id for the attribute in that predicate. In the descriptor search phase, we also process a request one predicate at a time. For each predicate, we determine the corresponding descriptor ids. We then create the Cartesian product of the descriptor ids of each predicate. Each result of that Cartesian product is a descriptor-id group. In the cluster search phase, we process a request using descriptor-id groups. For each descriptor-id group, we determine the corresponding cluster ids. Finally, in the address generation phase, we process a request using cluster ids. For each cluster id, we determine the corresponding secondary storage addresses of the records in that

cluster.

The processing during each phase of directory management is described in terms of the state and state-transition. Each state is represented by a rectangular box, which contains a description of the actions which take place in the state. The description of each phase will be given using a state transition diagram. These states include reading a particular type of data such as an attribute table node or waiting for concurrency control to grant a needed lock.

Lastly, to further simplify the discussion, we will not mention the waiting state. The waiting state occurs when an area of the primary memory, referred to as a buffer, is required for an I/O operation. Since there are only a finite number of buffers in the system, the waiting state is entered whenever a buffer is not available for the I/O operation.

4.1. The Attribute Search

The first phase of directory management is the attribute search. In this phase the attribute-id, if any, for the attribute in each predicate of the query is determined, as well as a pointer to the location of the descriptors in the DDIT for that attribute.

As described in [Boyn83], the attribute table is stored in a B-tree. A sample B-tree is in Figure 8. Processing is performed for each predicate in the query. Each node of the B-tree is stored in a different secondary storage location. Therefore, the nodes must be read and processed one at a time until the attribute is found. In addition, before the descriptor search can begin on a type-C attribute, that attribute must be locked by concurrency control (see Chapter 2 again).

The attribute search is described in more detail in Figure 9. Predicates are processed one at a time. For each predicate in the query the processing is as follows. First, the root node of the AT is read (marked with the number 1 in Figure 9). Then a sequence of nodes of the AT must be read; either the attribute is found or a leaf node is reached without finding the attribute (marked 2 in Figure 9). When the attribute is not in the AT, then we assume it is a non-directory attribute. In this case, no descriptor search is needed for that attribute, so the descriptor search for that predicate is finished by

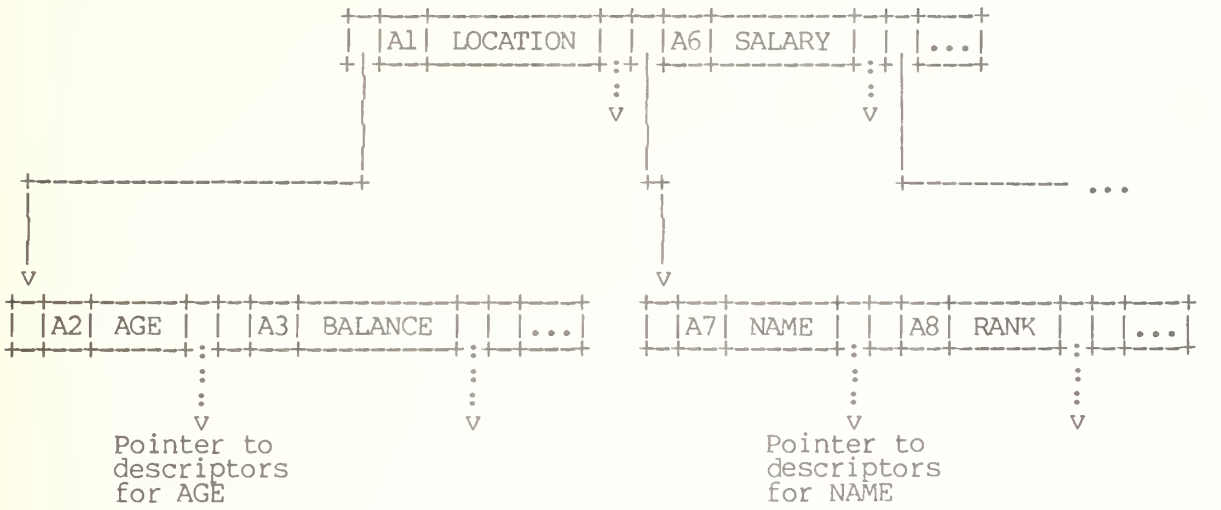
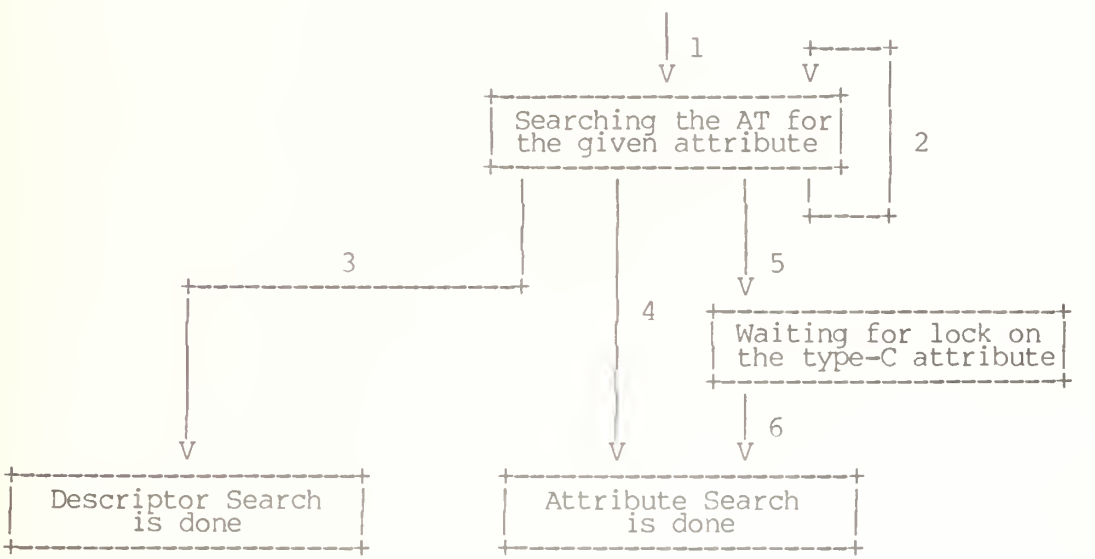


Figure 8. A Sample Attribute Table (AT)



Note: The procedure above is executed for each predicate in a query before the descriptor search can begin.

Figure 9. The Attribute Search for a Predicate in a Request

default(3). When the attribute is found in the AT, there are two possibilities. If the attribute is not a type-C attribute, then descriptor search can begin(4). However, if the predicate we are processing contains a type-C attribute, the attribute must be locked(5) before descriptor search can begin. In either case, when the attribute is found, the attribute id and the pointer to the descriptors for the attribute can now be obtained from the AT and made available to the next phase of directory management. We say that the attribute search is done for the attribute(6) and the descriptor search begins for the same attribute (see Figure 11).

The previous discussion focuses on the processing of one predicate of the query component. The attribute search phase processes all predicates of the query component, before the descriptor search phase for that request can begin. Thus, we can have an extra looping structure superimposed on the state diagram of Figure 9, which cycles through all predicates of the query component for a given request.

4.2. The Descriptor Search

The second phase of directory management is the descriptor search. In this phase the descriptor-ids corresponding to the predicate are determined. These descriptor-ids are stored in a B+tree as shown in the sample descriptor-to-descriptor-id table(DDIT) in Figure 10. Briefly, the DDIT consists of index nodes and sequence nodes. Index nodes are used to traverse the B+tree. Sequence nodes contain the information for a particular descriptor, e.g., the descriptor id, and the range of values for that id. Depending on the relational operator involved, the descriptor search first must determine either the leftmost sequence node, (for the operators, <, <=, NOT=), or an intermediate sequence node (for the operators, >, >=, =). If a range of values is required, then the descriptor search must follow the sequence nodes to determine the other descriptors. For an illustration, let us refer to Figure 10 and look at two examples. For the predicate AGE < 30, the descriptors D1, D2 and D3 must be determined. This is done by first retrieving the beginning sequence node, the one containing D1 and D2. Then the second sequence node must be examined to find D3. For the predicate 32 < AGE < 39, the descriptor corresponding to AGE = 32 must be determined. This descriptor is D4, which is in the second sequence node. Then D5 can be determined from the third sequence node.

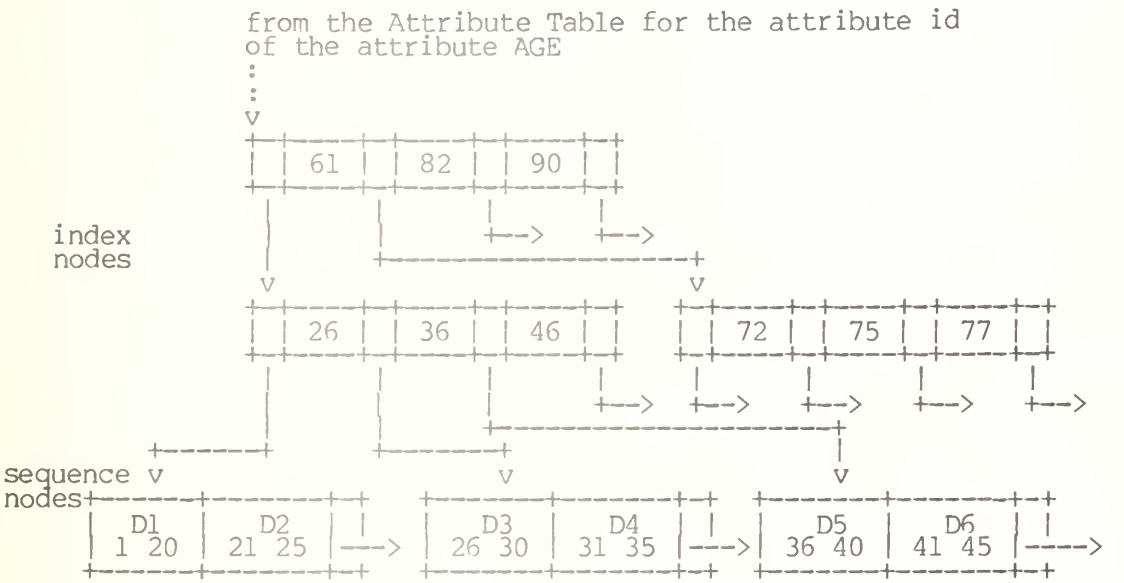


Figure 10. A Sample Descriptor-to-Descriptor-Id Table (DDIT)

The steps of descriptor search are shown in Figure 11. First the root node is read and processed(1). If there is only a root node for this attribute, i.e., the root node is a sequence node , then processing is finished(2). If the root node is not a sequence node, then the appropriate initial sequence node must be found. This will be the leftmost sequence node if the predicate relation is <, <= or NOT=. Otherwise, an intermediate node must be found.

In the first case the search is done by reading the leftmost child of the root node(3) and then continuing to read the leftmost child down the tree until the leftmost sequence node is found(4). If no additional sequence nodes are required, then the descriptor search is finished for this predicate(5). If additional sequence nodes are required, one(6) and possibly several(7) more sequence nodes are read. In the second case, i.e., an intermediate node must be found, a search down the B+tree is required(8,9). After the sequence node is found and if no additional sequence nodes are needed, the descriptor search is finished(10). Otherwise, additional sequence nodes are still required. One(11) and possibly several(7) more sequence nodes are read. After all the required sequence nodes have been read, the descriptor search is finished(12). At the end of this phase, the descriptor ids of the descriptors corresponding to the given predicate are found and made available to the next phase, the cluster search.

There is some additional processing if an insert request generates a new type-C subdescriptor. In this situation, the new descriptor-id must be received before this predicate is ready for the cluster search(13,14). After the descriptor-id is received, the predicate is ready for the cluster search(15). The actual insertion of the new descriptor-id is delayed until all descriptors have been determined. The steps required for the insertion of the new descriptor-id is described in Appendix D. If there is no new type-C subdescriptor, then no wait is necessary(12).

The above process is performed for each predicate of the query component of the request. At the end of the descriptor search phase, we have found a list of descriptor ids for each predicate of the query component. The Cartesian product of the lists of descriptor ids is formed, yielding a list of descriptor-id groups for the request. The descriptor-id groups are then passed to the third phase of directory management, the cluster search.

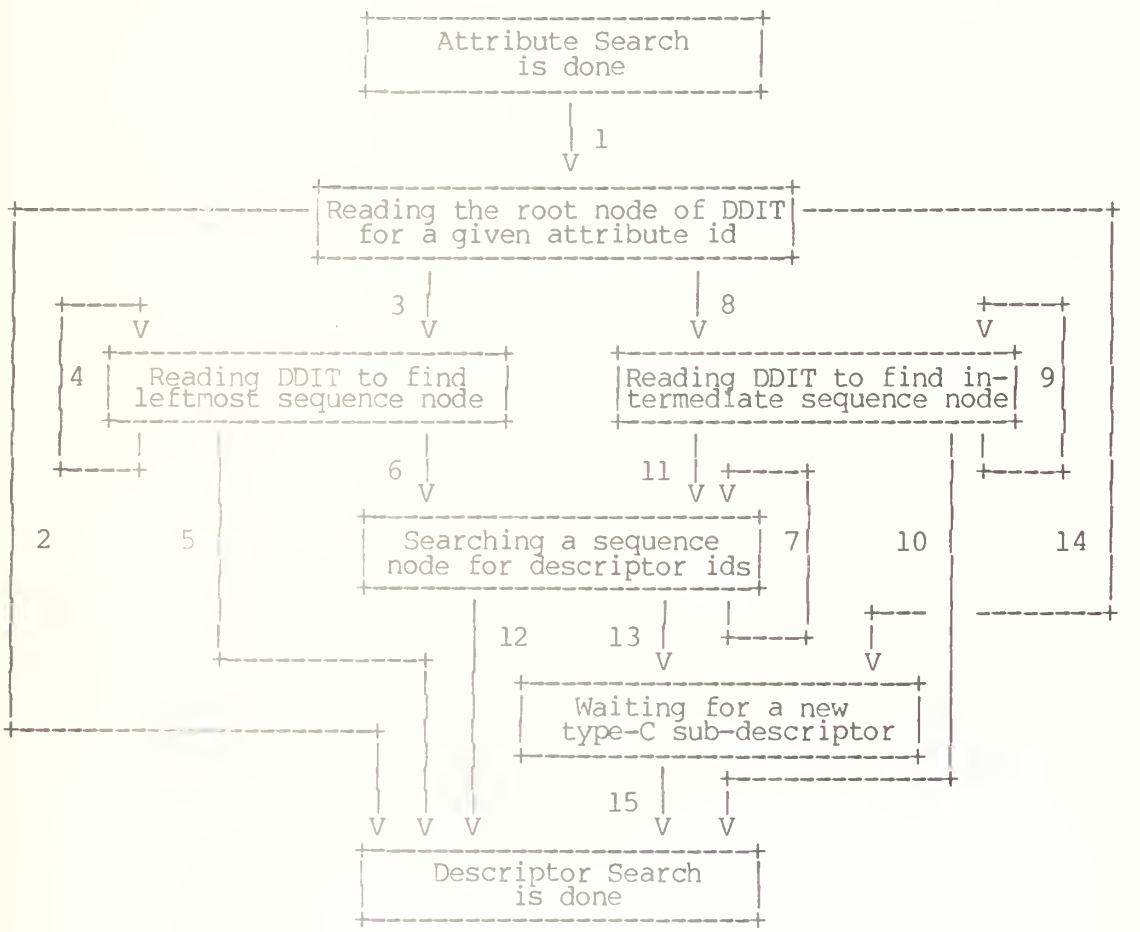


Figure 11. The Descriptor Search for a Predicate

4.3. The Cluster Search

The cluster ids of the clusters corresponding to the descriptor-id groups of the request must be determined by examining the cluster definition table(CDT). The CDT is stored in two parts, a Descriptor-Id-Cluster-Id-Bit-Map Table (DCBMT) and a Cluster-Id-to-Secondary-Storage-Address Table (CSSAT). The DCBMT is used during the cluster search phase, while the CSSAT is used during the address generation phase. A sample DCBMT is shown in Figure 12. Recall that a cluster is defined by a descriptor-id set. There is a bit map for each descriptor-id. Each bit map has one bit for each cluster-id in the database. A 1 bit corresponding to a cluster means that the descriptor-id appears in the definition of that cluster. Thus in the example, the given descriptor-id in the bit-map set defining clusters 2,6,11 and 17.

The bit-map index is used to find the bit-map set for a particular descriptor id. The bit-map index is stored in main memory. A bit-map set contains pointers to the first set of bits in the bit map for a group of descriptor ids. This bit map may be subdivided into several blocks. Thus, for each descriptor id, we retrieve one bit-map set and one or more bit-map blocks.

The cluster-ids corresponding to a descriptor-id group are determined by logically ANDing together the bit maps for each descriptor-id in the group. Thus input for the cluster search is the descriptor-id group. The states and transitions of the cluster search phase are shown in Figure 13.

The cluster search occurs in two steps. First the bit-map sets are determined for each descriptor id. Then the bit maps for each descriptor id are determined. At this point the bit maps for the descriptor-id groups are logically ANDed together to determine the required clusters. The bit-map sets are read first so as to avoid reading a bit-map set more than once.

With slightly more detail in Figure 13, the cluster search proceeds as follows. First the bit-map set is read for each descriptor-id(1). When all the bit-map sets have been read(2), the bit maps for each descriptor id can be found. The descriptor ids are again processed one at a time. The first block of bits from the bit map is read(4). Then any additional bits from the bit map are read(5). When all bits have been read, for every descriptor id, the cluster search is finished(6). If there is no bit map for this descriptor,

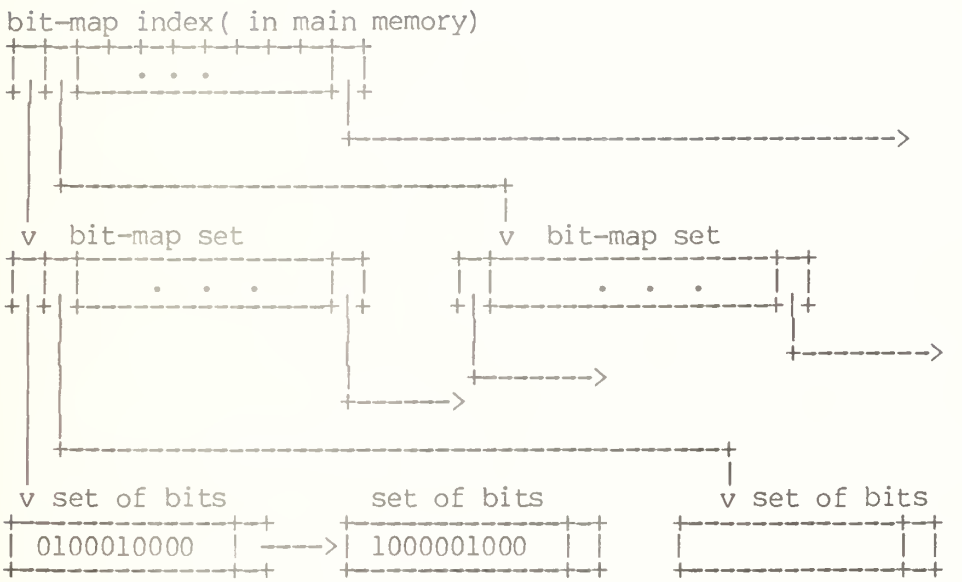


Figure 12. A Sample Descriptor-Id-Cluster-Id-Bit-Map Table(DCBMT)

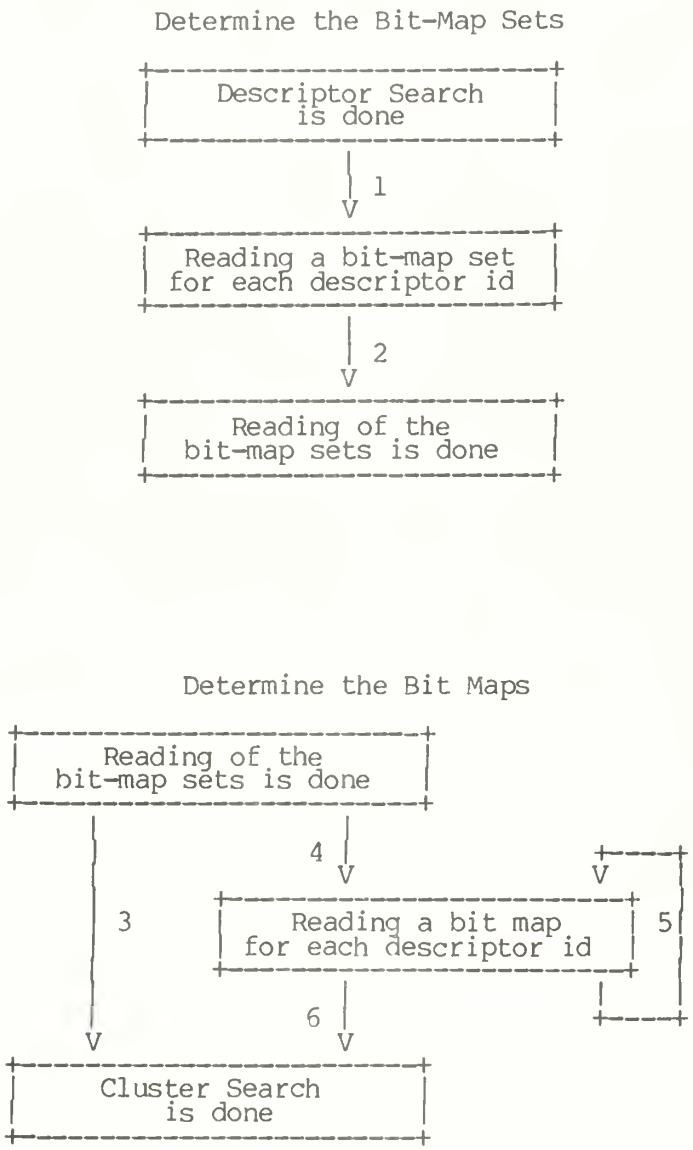


Figure 13. Cluster Search for Each Descriptor Id

then nothing is done(3).

We process each descriptor-id group of the query in the above manner to determine all of the clusters for the request. After the clusters have been determined by the cluster search and have been locked by concurrency control, it is time to determine the disk address(es) of the data records.

4.4. The Address Generation

This phase, the last phase of directory management, is called the address generation. As mentioned in the last section the disk addresses are stored in the Cluster-Id-to-Secondary-Storage-Address Table(CSSAT) as shown in the example in Figure 14. Each cluster has a fixed number of addresses stored in a cluster-address set, two in the example. Additional addresses are stored in an overflow area.

There are two cases to consider, the processing of insert requests and of non-insert, i.e., update, delete and retrieve, requests. For non-insert requests the disk addresses must be determined so that the appropriate data can be read by record processing. The CSSAT does not have to be modified. On the other hand, for insert requests it will be necessary to modify the CSSAT, either to increment the number of records in a track or to add a new track. Thus this case is more complex.

4.4.1. The Address Generation for a Non-insert Request

Let us first consider the case of a non-insert request. As with the cluster search, the address generation for a non-insert request is broken down into two steps for efficiency.

The states and transitions for each cluster are shown in Figure 15. In the first step, all the cluster-address sets are determined for each cluster(1,2). Then, the actual addresses are determined for each cluster. The determination of the addresses requires reading the first block of addresses(4) and possibly several overflow blocks(5). Processing for this cluster is finished when there are no additional addresses to be found(6). If there are no records for the cluster in this backend, then, of course, no reading is required(3).

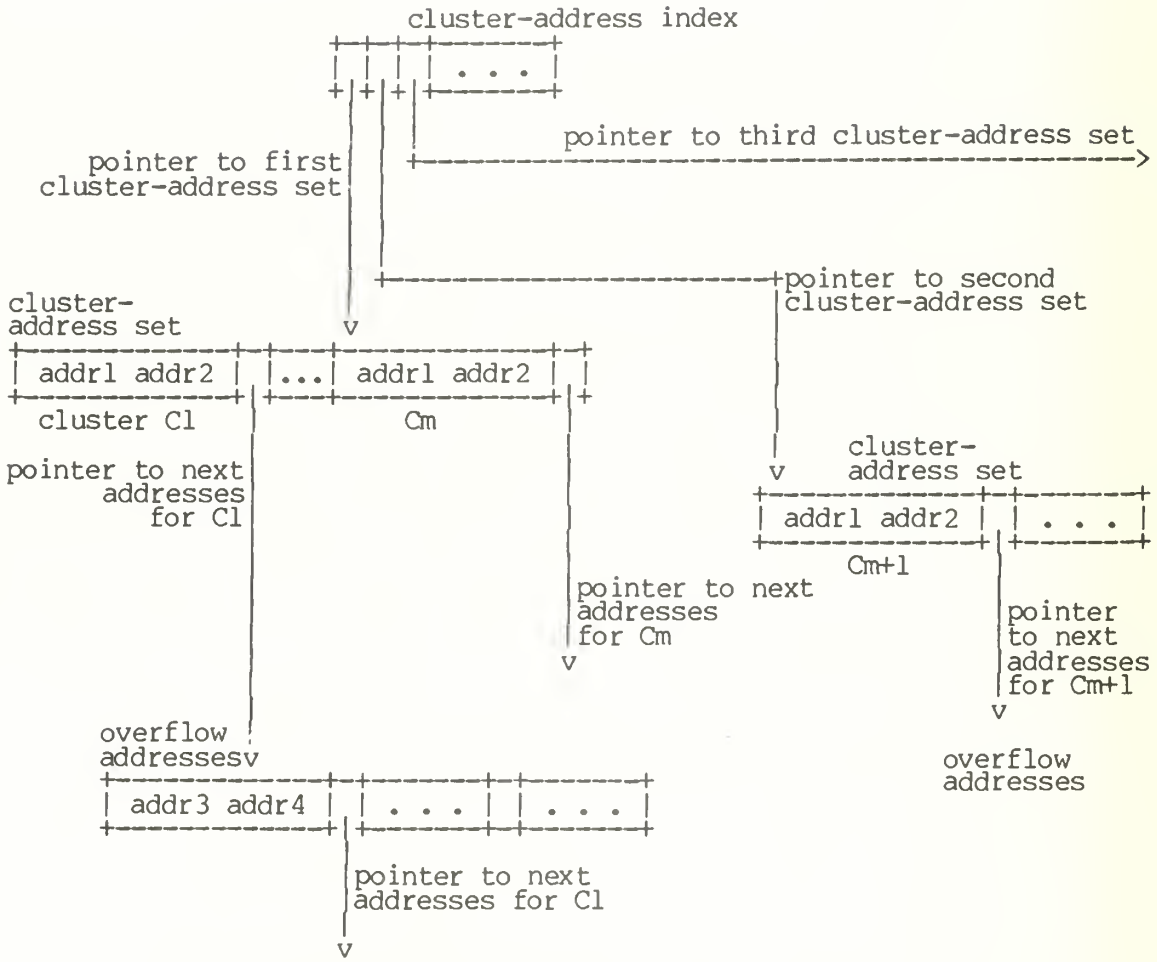


Figure 14. A Sample Cluster-Id-to-Secondary-Storage-Address Table (CSSAT)

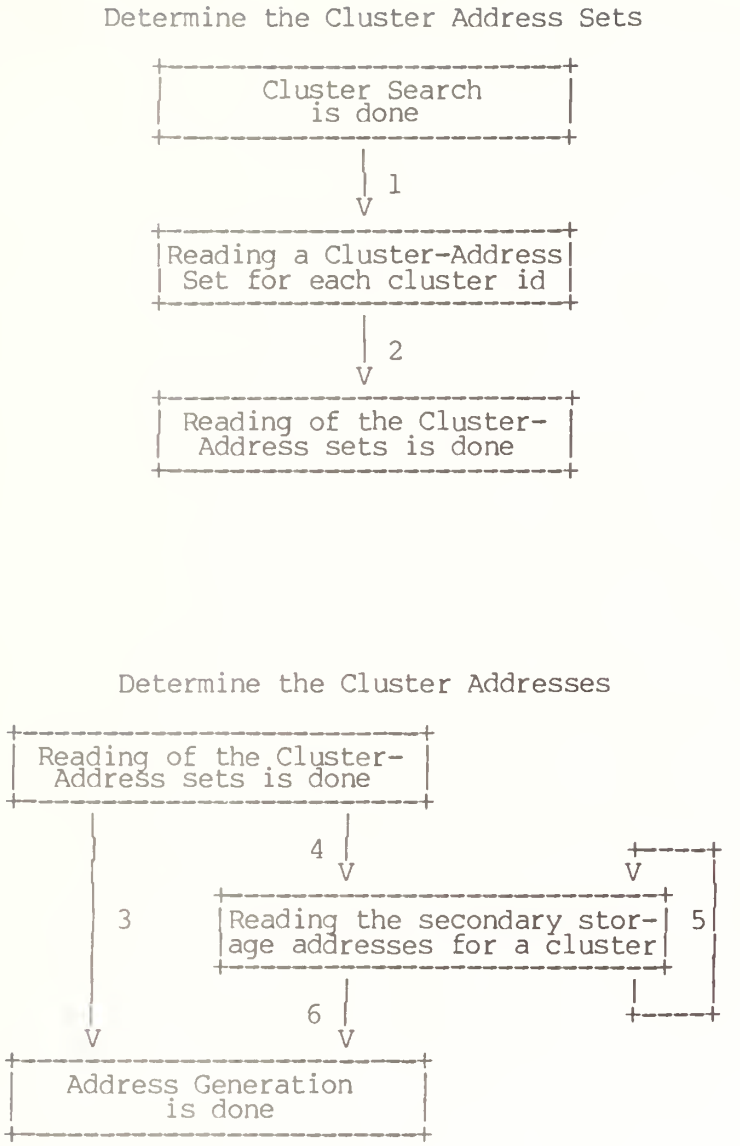


Figure 15. Address Generation (non-insert request) for Each Cluster Id

4.4.2. The Address Generation for an Insert Request

When processing an insert request, there is only one cluster to be determined. In fact, only one address at which to store the record needs to be determined. However, the CSSAT must be updated to reflect the insertion of the new record. Reading of the CSSAT is similar to the non-insert case. However additional processing is needed to update the CSSAT. The states and transitions for each cluster are shown in Figure 16.

Let us first consider the case where a new cluster is being created. We get a track for the new cluster. If no cluster-address set exists, then one is created, the new track address is inserted and the cluster-address set is written to secondary storage(1). On the other hand, if the cluster-address set already exists, it must be read(2). The new track address is inserted and the updated cluster-address set written(3). In either case, the address generation phase is done(4).

Next let us consider the case of an old cluster. In this case an existing cluster-address set must be read(2). At this point, processing differs depending on whether or not overflow address blocks must be processed. No overflow processing is required if the new record fits in the last track assigned to the cluster. In this case, the remaining space in the track is updated and the cluster-address set is written(3). A second case also requires no processing of overflow blocks. If a new data track is required and there is room in the cluster-address set for the new track address, then this address is added, the space remaining in the track updated and the cluster-address set is written(3). In either case, the address generation phase is done after the cluster-address set has been written(4).

The most complex processing is required when it is necessary to use overflow blocks. Such processing occurs in three cases. The simplest of these cases occurs when it is necessary to create the first overflow block. In this case the address of the overflow block, the address of the new track, and the space remaining in the new track are added to the cluster-address set, which is then written(5). The new address is also put in the overflow block and that block is written(6). The address generation phase is done(13) when the write is finished.

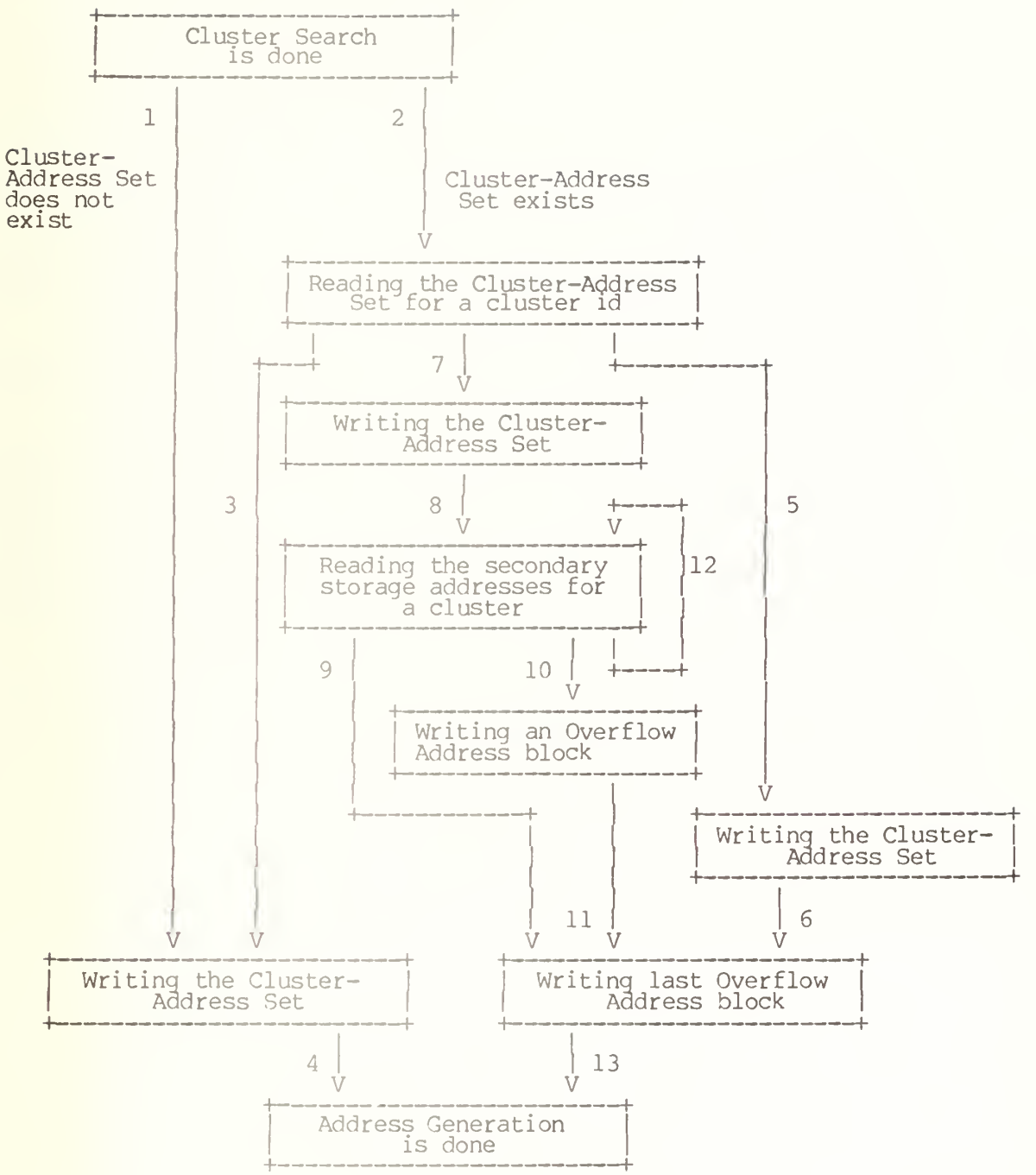


Figure 16. Address Generation (insert request)

The final two cases occur when the last track is full and the cluster in question already has one or more overflow address blocks. Since the last track is full, a new track is required and the address of that track must be added to the end of the overflow addresses of the CSSAT. Processing begins by reading the first overflow address(8). If other overflow addresses are present, they must also be read(12). If there is room in the last overflow block., the new address is added to the overflow block and that block is written to the secondary storage(9). On the other hand, there may be no room in the last overflow block. In this case, the address for the new block is obtained and a pointer to it stored in the previous last block(10). After that write is finished, the new overflow addresses may be written(11). In either case, the address generation phase is done after the last overflow block has been written(13).

In all of the discussion above, it is important to remember that address generation for an insert request occurs at only one backend, the one the controller has chosen to actually insert the new record. The other backends are finished processing the insert after they have determined the cluster for the insert and the controller has broadcast the number of the backend which is to store the new record.

5. AN UPDATED DESCRIPTION OF MDBS MESSAGES

In this chapter we examine the revisions made to the MDBS message passing facilities first described in [Boyn83]. In the MDBS message passing facilities there are 31 message types and one general message format (shown in Figure 17). This same format is used for each of the three message passing facilities, namely, messages within the controller, messages with the backend, and messages between computers. Messages between computers are divided into two classes, messages between backends, and messages between the controller and the backends. Figure 18 describes each of the MDBS message types.

Communication between computers in MDBS is achieved by using a time-division-multiplexed bus called the parallel communication link (PCL) [DEC79a]. We built a software interface to this bus for each computer consisting of two complimentary processes. The first process, get_pcl, gets messages from other computers off the PCL. The second process, put_pcl, puts messages on the bus to be sent to other computers. The controller and each backend have their own get_pcl and put_pcl processes.

In the rest of this chapter, we first present the revised list of MDBS message definitions. Then, we examine the sequence of actions for an insert, delete, retrieve and update request in the MDBS message passing environment.

Message Type	(a numeric code).
Message Sender	(a numeric code).
Message Receiver	(a numeric code).
Message Text	(an alphanumeric field terminated by an end of message marker).

Figure 17. MDBS General Message Format

	MESSAGE-TYPE NUMBER AND NAME	SRC	DEST	PATH
1	TRAFFIC UNIT	HOST	REQP	HC
	REQUEST RESULTS	PP	HOST	CH
	NUMBER OF REQUESTS IN A TRANSACTION	REQP	PP	C
	AGGREGATE OPERATORS	REQP	PP	C
5	REQUESTS WITH ERRORS	REQP	PP	C
	PARSED TRAFFIC UNIT	REQP	DM	CB
	NEW DESCRIPTOR ID	IIG	DM	CB
	BACKEND NUMBER	IIG	DM	CB
	CLUSTER ID	DM	IIG	BC
10	REQUEST FOR NEW DESCRIPTOR ID	DM	IIG	BC
	BACKEND RESULTS FOR A REQUEST	RECP	PP	BC
	BACKEND AGGREGATE OPERATOR RESULTS	RECP	PP	BC
	RECORD THAT HAS CHANGED CLUSTER	RECP	REQP	BC
	RESULTS OF A RETRIEVE OR FETCH CAUSED BY AN UPDATE	RECP	REQP	BC
15	DESCRIPTOR IDS	DM	DMs	BB
	REQUEST AND DISK ADDRESSES	DM	RECP	B
	CHANGED CLUSTER RESPONSE	DM	RECP	B
	FETCH	DM	RECP	B
	OLD AND NEW VALUES OF ATTRIBUTE BEING MODIFIED	RECP	DM	B
20	TYPE-C ATTRIBUTES FOR A TRAFFIC UNIT	DM	CC	B
	DESC-ID GROUPS FOR A TRAFFIC UNIT	DM	CC	B
	CLUSTER IDS FOR A TRAFFIC UNIT	DM	CC	B
	RELEASE ATTRIBUTE	DM	CC	B
	RELEASE ALL ATTRIBUTES FOR AN INSERT	DM	CC	B
25	RELEASE DESCRIPTOR-ID GROUPS	DM	CC	B
	ATTRIBUTE LOCKED	CC	DM	B
	DESCRIPTOR-ID GROUPS LOCKED	CC	DM	B
	CLUSTER IDS LOCKED	CC	DM	B
29	NO MORE GENERATED INSERTS	RECP	REQP	BC
29	NO MORE GENERATED INSERTS	REQP	DM	CB
29	NO MORE GENERATED INSERTS	DM	RECP	BC
30	REQUEST ID OF A FINISHED REQUEST	RECP	CC	B
31	AN UPDATE REQUEST HAS FINISHED	RECP	DM	B
31	AN UPDATE REQUEST HAS FINISHED	DM	CC	B

SOURCE OR DESTINATION DESIGNATION		PATH DESIGNATION	
HOST	: HOST MACHINE (TEST-INT)	H	: HOST
REQP	: REQUEST PREPARATION	C	: CONTROLLER
IIG	: INSERT INFORMATION GENERATION	C	: CONTROLLER
PP	: POST PROCESSING	C	: CONTROLLER
DM	: DIRECTORY MANAGEMENT	B	: A BACKEND
RECP	: RECORD PROCESSING	B	: A BACKEND
CC	: CONCURRENCY CONTROL	B	: A BACKEND

Figure 18. The MDBS Message Types

5.1. Revised Definitions of MDBS Messages

In this section we give short descriptions of the revised definitions of MDBS messages. The first group of messages are those between the host and the controller and within the controller itself. These messages are shown in Figure 19.

Message type : (1) Host Traffic Unit
Source : Host
Destination : Request Preparation
Explanation : The traffic unit represents a single request or transaction from a user at the host machine.

Message type : (2) Request Results
Source : Post Processing
Destination : Host
Explanation : Contains the results for a request after being collected from all the backends and aggregated if necessary.

Message type : (3) Number of Requests in a Transaction
Source : Request Preparation
Destination : Post Processing
Explanation : Request Preparation sends to Post Processing the number of requests in a traffic unit. This enables Post Processing to determine whether the processing of a traffic unit is complete.

Message type : (4) Aggregate Operators
Source : Request Preparation
Destination : Post Processing
Explanation : Request Preparation sends the aggregate operators to Post Processing.

Message type : (5) Requests with Errors
Source : Request Preparation
Destination : Post Processing
Explanation : Requests with errors will be found in Request Preparation by the Parser and sent to the Post Processing directly. Post Processing will send the requests with errors back to the host.

The next set of messages deals with the communication between the controller and the Directory Management process within each backend. These messages can be found in Figure 20.

Message type : (6) Parsed Traffic Unit
Source : Request Preparation
Destination : Directory Management
Explanation : This is the prepared traffic unit sent by Request Preparation.

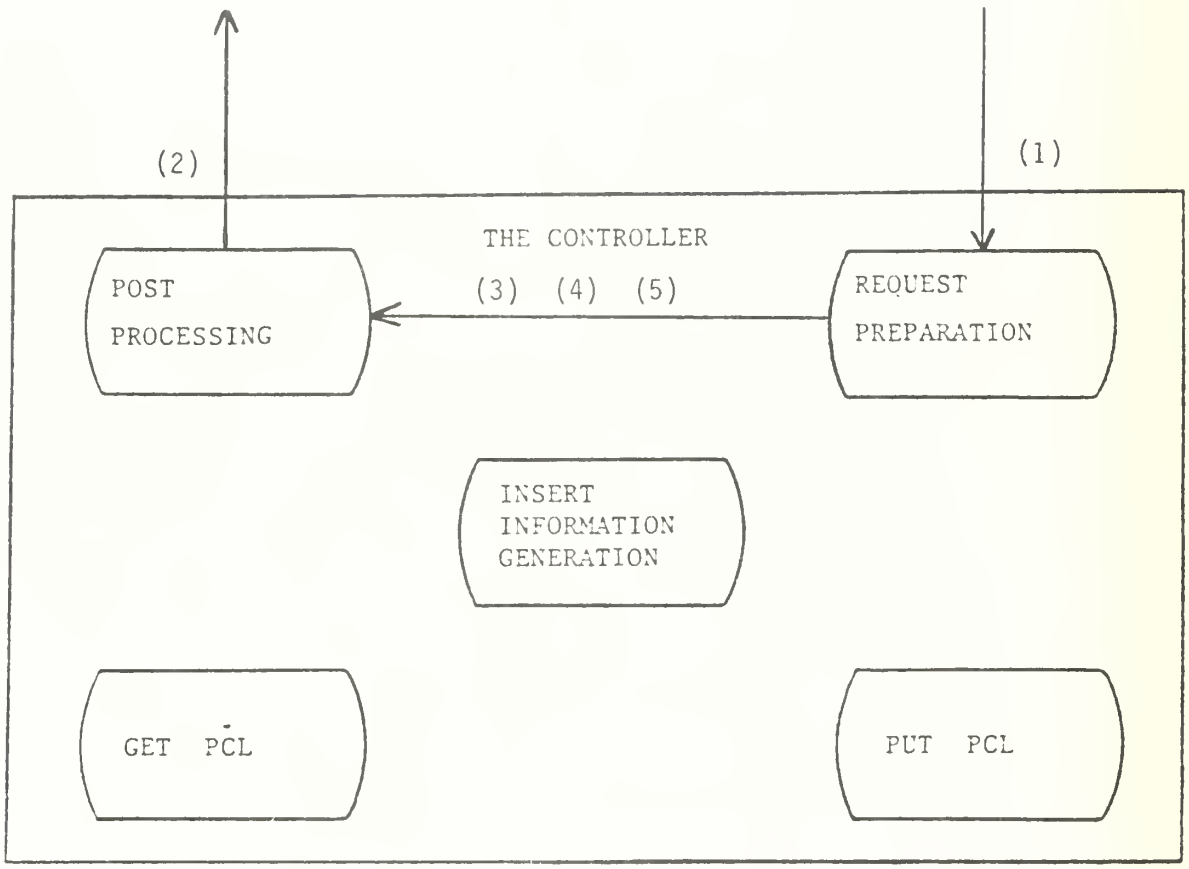


Figure 19. Controller Related Messages

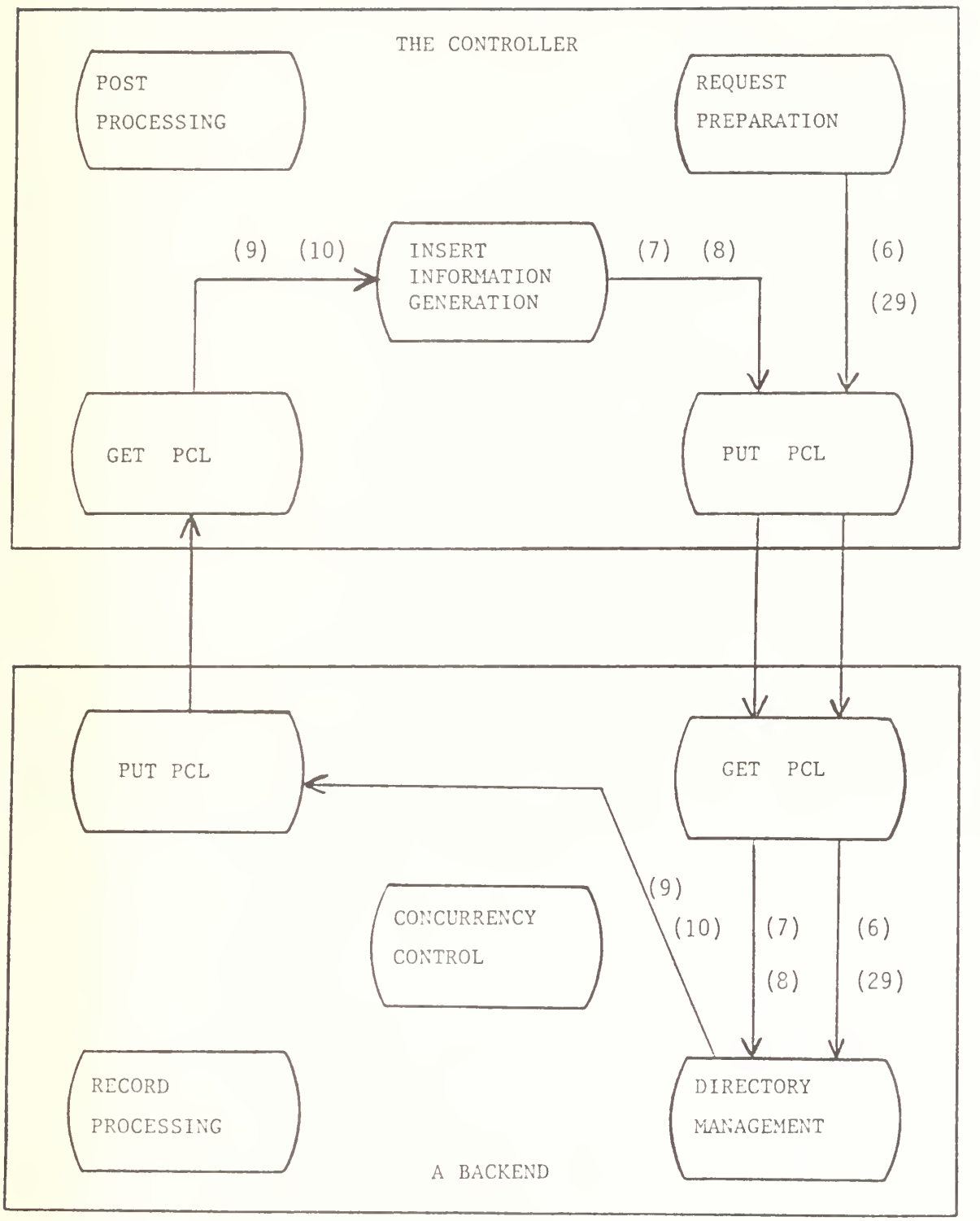


Figure 20. REQP, IIG (Controller), and DM (Backend) Related Messages

Message type : (29) No More Generated Inserts
Source : Request Preparation
Destination : Directory Management
Explanation : This message indicates that insert request for all the records that have changed cluster as a result of an update request have been generated and sent to Directory Management.

Message type : (7) New Descriptor Id
Source : Insert Information Generation
Destination : Directory Management
Explanation : This message is a response to the Directory Management request for a new descriptor id.

Message type : (8) Backend Number
Source : Insert Information Generation
Destination : Directory Management
Explanation : This message is used to specify which backend is to insert a record.

Message type : (9) Cluster Id
Source : Directory Management
Destination : Insert Information Generation
Explanation : Directory Management sends a cluster id to Insert Information Generation for an insert request. IIG will decide where to do the insert.

Message type : (10) Request for New Descriptor Id
Source : Directory Management
Destination : Insert Information Generation
Explanation : When Directory Management has found a new descriptor it is sent to Insert Information Generation to generate an id.

The third group of messages deal with the flow from the Record Processing process in a backend to the Post Processing and Request Preparation processes in the controller. Figure 21 shows the flow of these messages.

Message type : (11) Results of a Request from a Backend
Source : Record Processing
Destination : Post Processing
Explanation : This message contains the results that a specific backend found for a request.

Message type : (12) Aggregate Operator Results from a Backend
Source : Record Processing
Destination : Post Processing
Explanation : When an aggregate operation needs to be done on the retrieved records, each backend will do as much aggregation as possible in the aggregate operation function of Record Processing. This message carries those results to Post Processing.

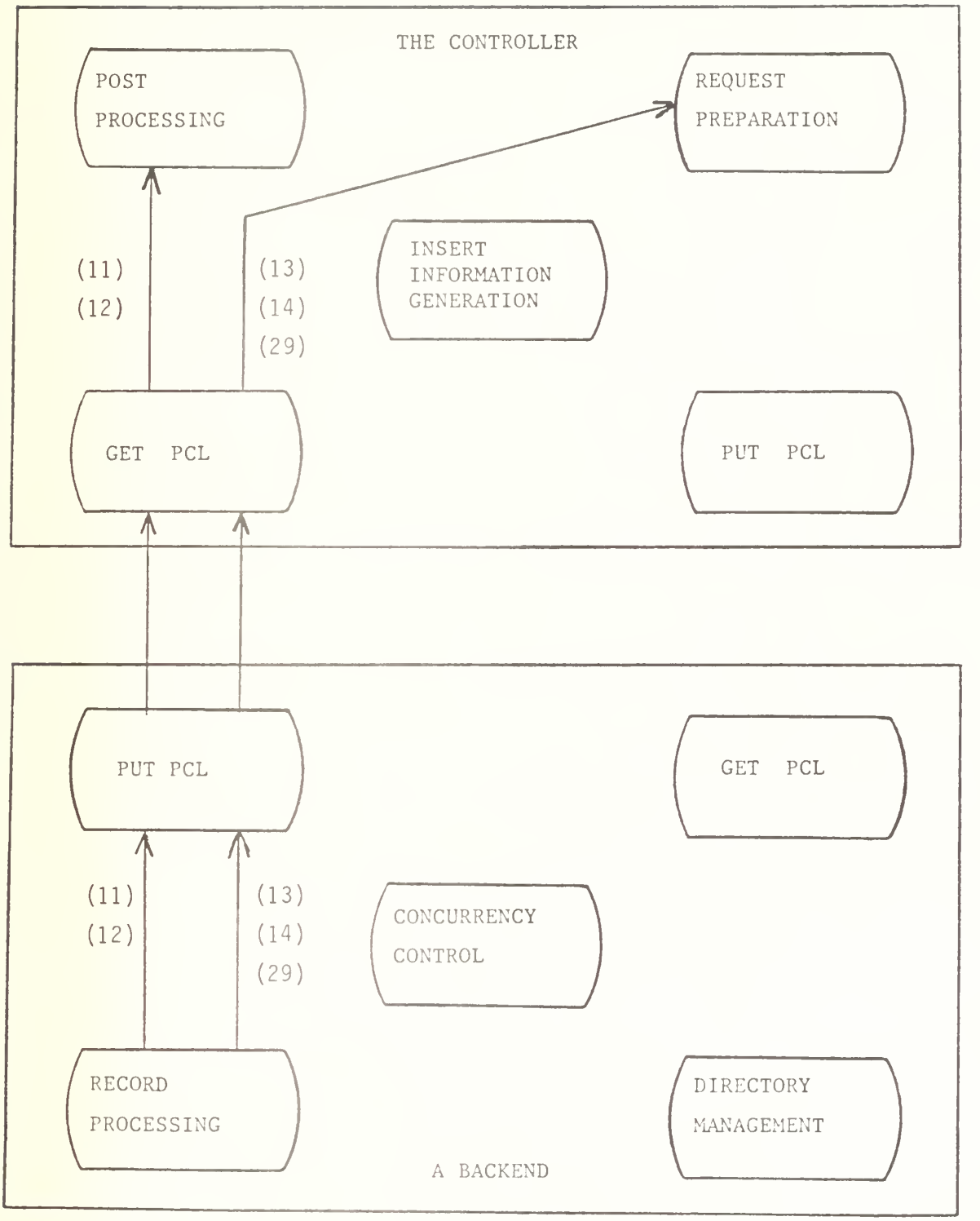


Figure 21. REQP, RECP and PP Related Messages

Message type : (13) Record that has Changed Cluster
Source : Record Processing
Destination : Request Preparation
Explanation : This message is a record which has changed cluster, Request Preparation will prepare it as an insertion and send it to the backends.

Message type : (29) No More Generated Inserts
Source : Record Processing
Destination : Request Preparation
Explanation : This message indicates that all the records that have changed cluster as a result of an update request have been sent to Request Preparation.

Message type : (14) Results of a Retrieve or Fetch Caused by an Update
Source : Record Processing
Destination : Request Preparation
Explanation : This message carries the information from a retrieve or fetch back to Request Preparation to complete an update with type-III or type-IV modifier.

The following descriptions are for messages between Directory Management processes residing on different backends and between Directory Management and Record Processing within a backend. These messages are shown in Figure 22.

Message type : (15) Descriptor Ids
Source : Directory Management
Destination : Directory Management (other backends)
Explanation : This message contains the results of descriptor search by Directory Management.

Message type : (16) Request and Disk Addresses
Source : Directory Management
Destination : Record Processing
Explanation : This message contains a request and disk addresses for Record Processing to come up with the results for the request.

Message type : (17) Changed Cluster Response
Source : Directory Management
Destination : Record Processing
Explanation : Directory Management uses this message to tell Record Processing whether an updated record has changed cluster.

Message type : (29) No More Generated Inserts
Source : Directory Management
Destination : Record Processing
Explanation : This message indicates that all insert requests generated as a result of an update request have been sent to Record Processing.

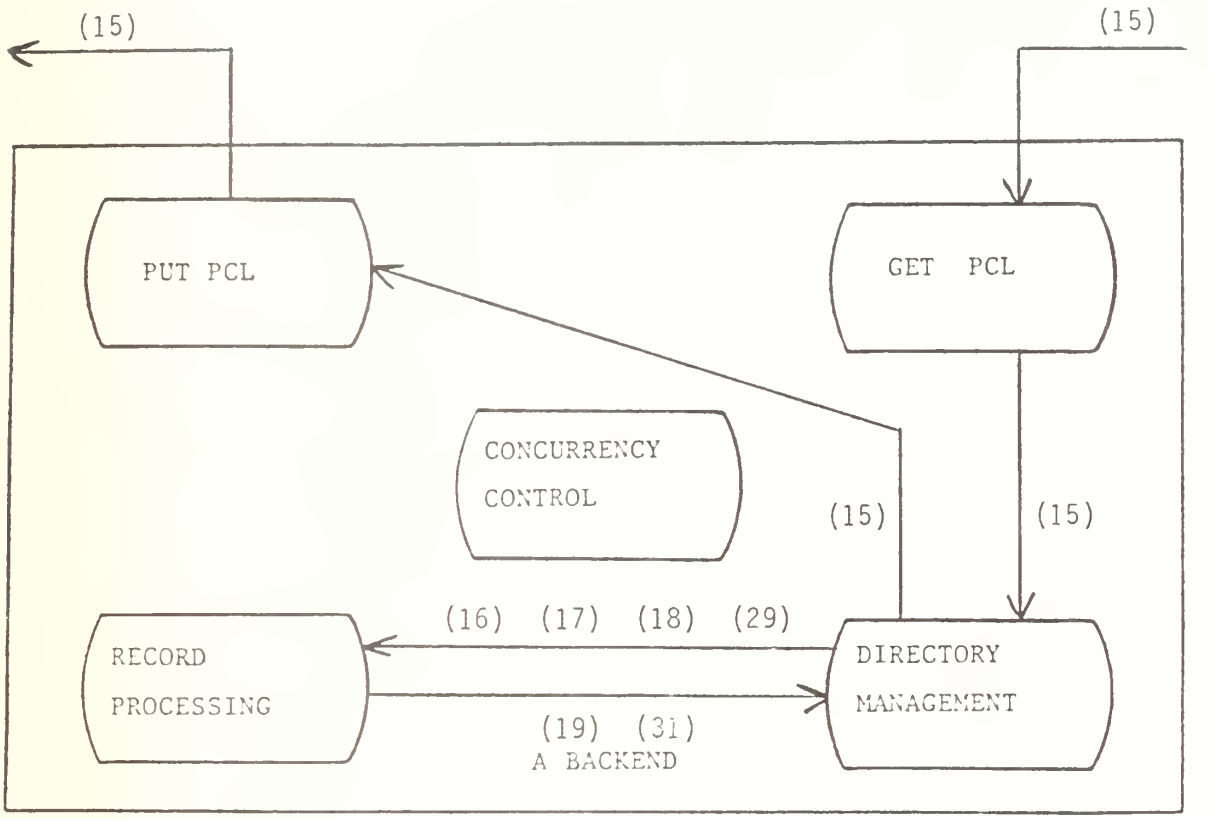


Figure 22. DM and RECP Related Messages

Message type : (18) Fetch
Source : Directory Management
Destination : Record Processing
Explanation : Fetch is a special retrieval of information for Request Preparation due to an update request with type-IV modifier.

Message Type : (19) Old and New Values of Attribute being Modified
Source : Record Processing
Destination : Directory Management
Explanation : Record Processing uses this message to check whether a record that has been updated has changed cluster.

Message Type : (31) An Update Request has Finished
Source : Record Processing
Destination : Directory Management
Explanation : Record Processing signals Directory Management that an update request has finished execution.

The last set of messages are the Concurrency Control related messages. These messages pass information from either Directory Management or Record Processing to Concurrency Control. These are shown in Figure 23.

Message Type : (20) Type-C Attributes for a Traffic Unit
Source : Directory Management
Destination : Concurrency Control
Explanation : Concurrency Control takes the attributes in this message and determines when Descriptor Search for an attribute can be performed.

Message Type : (21) Descriptor-id Groups for a Traffic Unit
Source : Directory Management
Destination : Concurrency Control
Explanation : Concurrency Control takes the descriptor-id groups in this message and determines when Cluster Search for a request can be performed.

Message Type : (22) Cluster Ids for a Traffic Unit
Source : Directory Management
Destination : Concurrency Control
Explanation : Concurrency Control takes the cluster ids in this message and determines when a request can continue with Address Generation and the rest of request execution.

Message Type : (23) Release Attribute
Source : Directory Management
Destination : Concurrency Control
Explanation : Directory Management uses this message to signal Concurrency Control that a request has performed Descriptor Search on an attribute, and the lock on the attribute held by the request can be released.

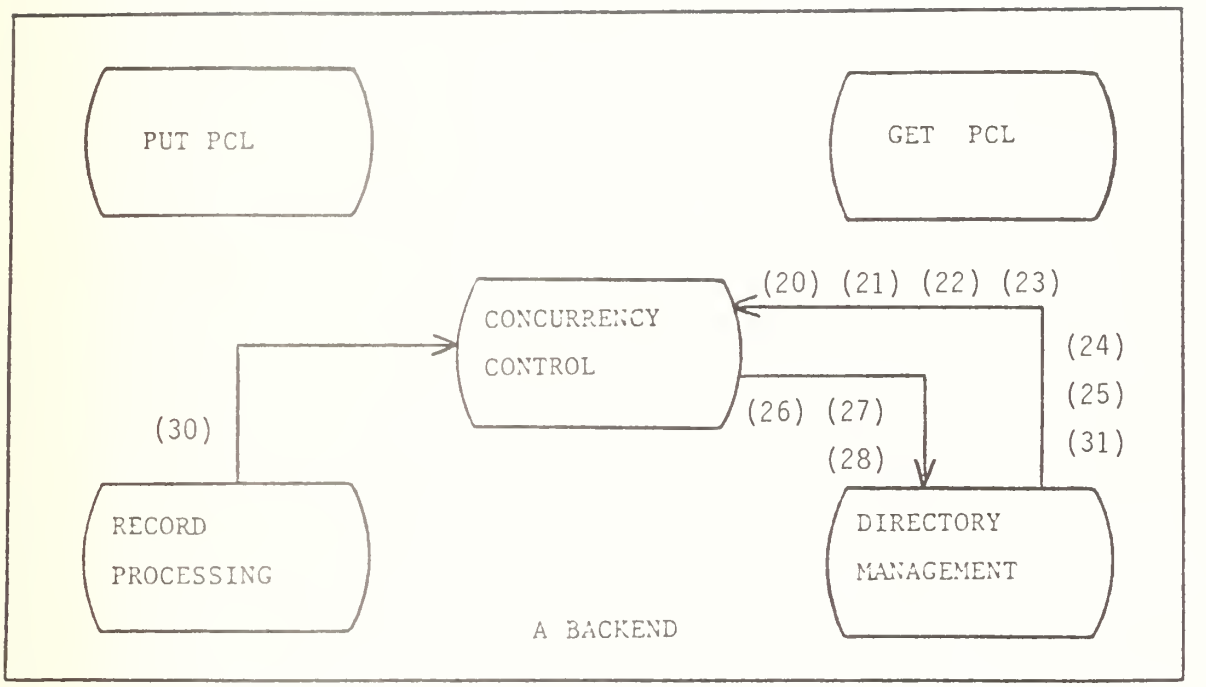


Figure 23. DM, RECP, and CC Related Messages

Message Type : (24) Release All the Attributes for an Insert
Source : Directory Management
Destination : Concurrency Control
Explanation : Directory Management uses this message to signal Concurrency Control that an insert request has performed Descriptor Search on all the attributes, and the locks on the attributes held by the request can be released.

Message Type : (25) Release Descriptor-Id Groups
Source : Directory Management
Destination : Concurrency Control
Explanation : Directory Management uses this message to signal Concurrency Control that an insert request has performed Cluster Search for a request, and the locks on the descriptor-id groups held by the request can be released.

Message Type : (31) An Update Request Has Finished
Source : Directory Management
Destination : Concurrency Control
Explanation : Directory Management uses this message to signal Concurrency Control that an update request has finished execution, and all the locks held by the request can be released.

Message Type : (26) Attribute Locked
Source : Directory Management
Destination : Concurrency Control
Explanation : Concurrency Control signals Directory Management that an attribute is locked for a request, and Descriptor Search can be performed.

Message Type : (27) Descriptor-Id Groups Locked
Source : Directory Management
Destination : Concurrency Control
Explanation : Concurrency Control signals Directory Management that the Descriptor-id groups needed by a request are locked, and Cluster Search can be performed.

Message Type : (28) Cluster Ids Locked
Source : Directory Management
Destination : Concurrency Control
Explanation : Concurrency Control signals Directory Management that the cluster ids needed by a request can continue with address Generation and the rest of request execution.

Message Type : (23) Request Id of a Finished Request
Source : Record Processing
Destination : Concurrency Control
Explanation : Record Processing signals Concurrency Control that a non-update request has finished execution, and the locks on cluster ids held by the request can be released.

5.2. Request Execution in MDBS - Viewed Vie Message Passing

In this section, we describe the sequence of actions for a request as it moves through MDBS. The sequence of actions will be described in terms of the types of messages passed between the MDBS processes: Request Preparation (REQP), Insert Information Generation (IIG), Post Processing (PP), Directory Management (DM), Record Processing (RECP) and Concurrency Control (CC). The order in which messages are passed will be denoted alphabetically ('a' is first). The digit following the ordering letter will be the message number as shown in Figure 18. We examine the four types of requests, insert, delete, retrieve, and update.

5.2.1. Sequence of Actions for an Insert Request

The sequence of actions for an insert request is shown in Figure 24. The traffic unit (a1) comes into REQP from the host carrying an insert request. REQP sends to PP the number of requests in the traffic unit (b3). After preparation, the formatted request is sent to DM from REQP (c6). DM sends the type-C attributes needed by the request to CC (d20). Once an attribute is locked, and Descriptor Search can be performed, CC signals DM (e26). DM will then perform Descriptor Search. From DM, descriptor ids for the request will be sent to the other backends in the MDBS system (f15). DM also signals CC to release the locks on attributes (g24). The descriptor ids found by the other backends will be received by DM (h15). DM now sends the descriptor-id group for the request to CC (i21). Once the descriptor-id group is locked and Cluster Search can be performed, CC signals DM (j27). DM will then perform Cluster Search. To determine where the insert will occur, DM will send the insert cluster id to IIG (k9). Once the backend has been selected, IIG will send the backend number to DM (m8). DM updates its directory tables if needed, and signals CC to release the lock held by the request on the descriptor-id group (n25). DM will send the insert cluster id to CC (o22). CC will respond to DM when the insert request can proceed with Address Generation and the rest of request execution (p28). With the go ahead from CC, DM will perform Address Generation and send RECP the request and its required disk address (q16). After the insert has occurred, RECP will notify CC that the request is done (r30), followed by a message to PP that the request has completed (s11). PP will finish the processing by sending a results message to the host (t2).

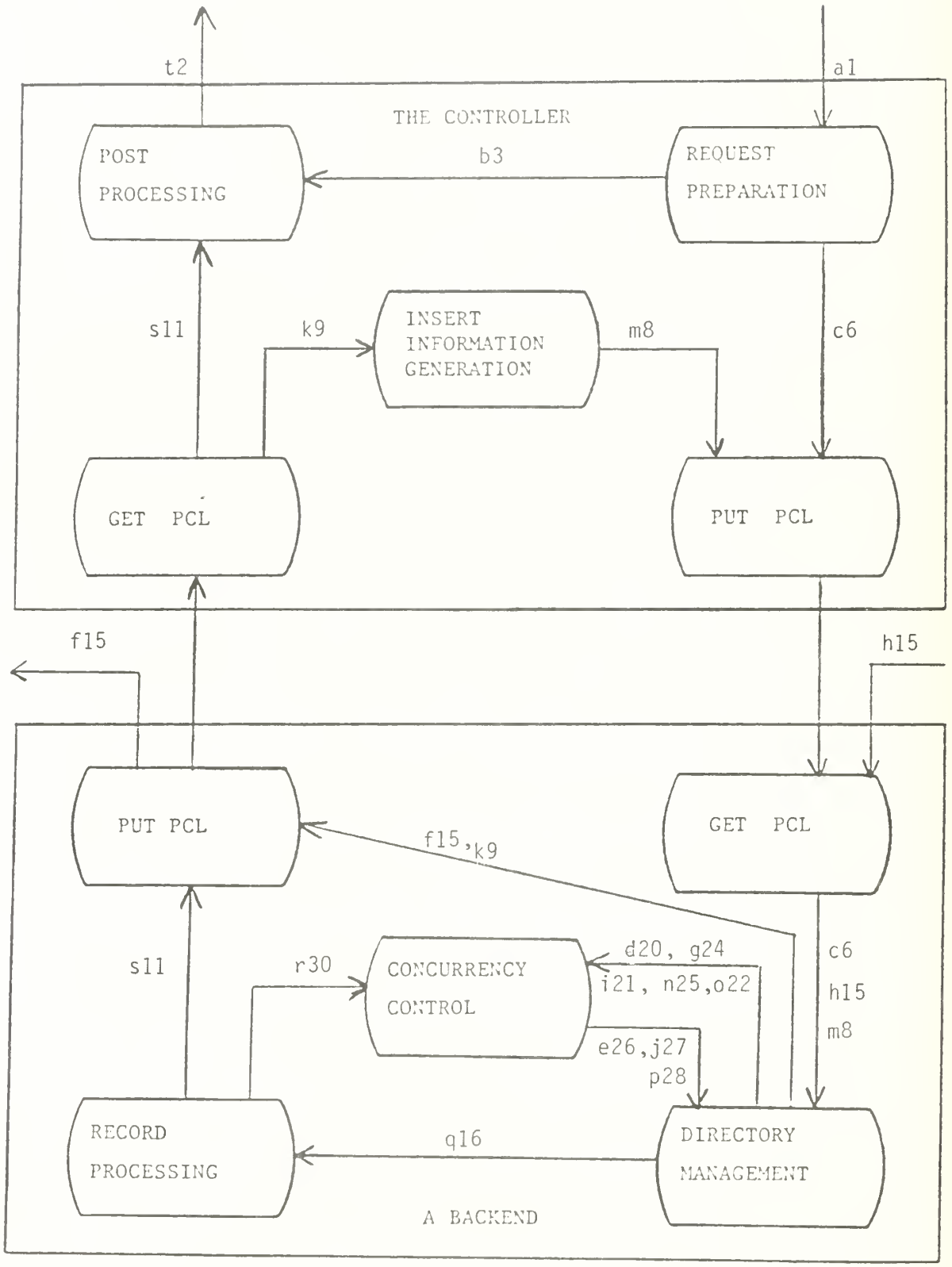


Figure 24. Sequence of Messages for an Insert Request

5.2.2. Sequence of Actions for a Delete Request

The sequence of actions for a delete request is shown in Figure 25. A traffic unit is sent to REQP from the host containing the delete request (a1). REQP notifies PP of the number of requests in the traffic unit (b3). Next, REQP sends the request down to DM (c6). DM sends the type-C attributes needed by the request to CC(d20). Once an attribute is locked and Descriptor Search can be performed, CC signals DM (e26). DM performs Descriptor Search and signals CC to release the lock on the attribute (f23). The descriptor ids for the request are next sent to the other backends from DM (g15). The other backends respond with the descriptor ids they have found (h15). DM sends the descriptor-id groups for the request to CC (i21). Once the descriptor-id groups are locked and Cluster Search can be performed, CC signals DM (j27). DM performs Cluster Search and signals CC to release the locks on the descriptor-id groups (k25). DM will next send the cluster ids for the delete request to CC (m22). Once the cluster ids are locked and the request can continue with Address Generation and the rest of request execution, CC signals DM (n28). DM will then perform Address Generation and send to RECP the address and the request (p16). After RECP has performed the delete request, it will notify CC that the request is through (p30). PP will then receive a results message from RECP telling it that the request is done (q11). PP will then notify the host with a results message (r2).

5.2.3. Sequence of Actions for a Retrieve Request with Aggregate Operator

The sequence of actions for a retrieve request is shown in Figure 26. First the retrieve request comes to REQP from the host (a1). REQP sends two messages to PP: the number of requests in the transaction (b3) and the aggregate operator of the request (c4). The third message sent by REQP is the parsed traffic unit which goes to DM in the backends (d6). DM sends the type-C attributes needed by the request to CC (e20). Once an attribute is locked and CC can be performed, CC signals DM (f26). DM will then perform Descriptor Search and signal CC to release the lock on that attribute (g23). DM will send the descriptor ids for the request to the other backends (h15). The DM processes in the other backends will send their descriptor ids to the DM process residing in this backend (i15). DM now sends the descriptor-id groups for the retrieve request to CC (j21). Once the descriptor-id groups are locked and Cluster Search can be performed, CC signals DM (k27). DM will

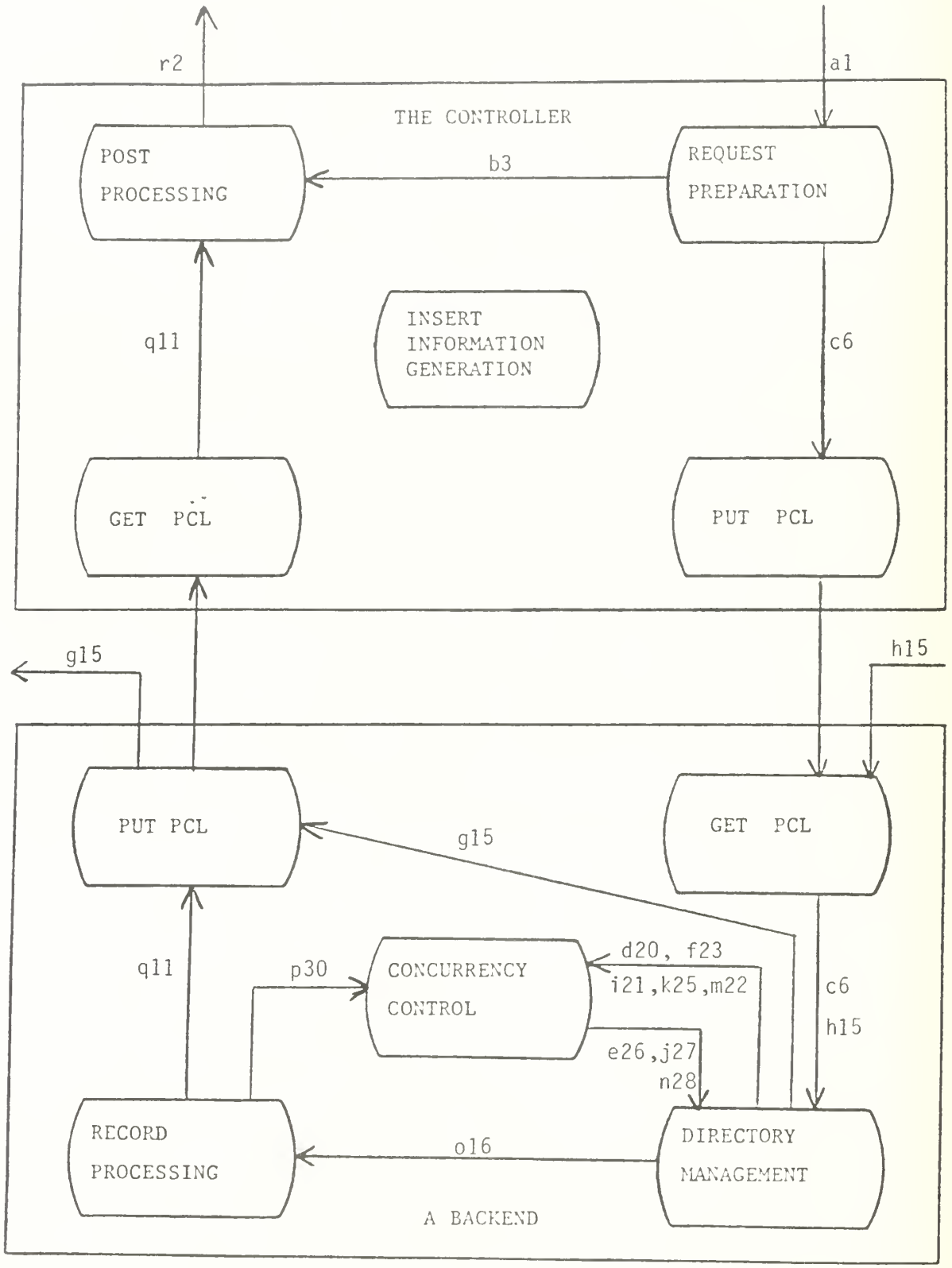


Figure 25. Sequences of Messages for a Delete Request

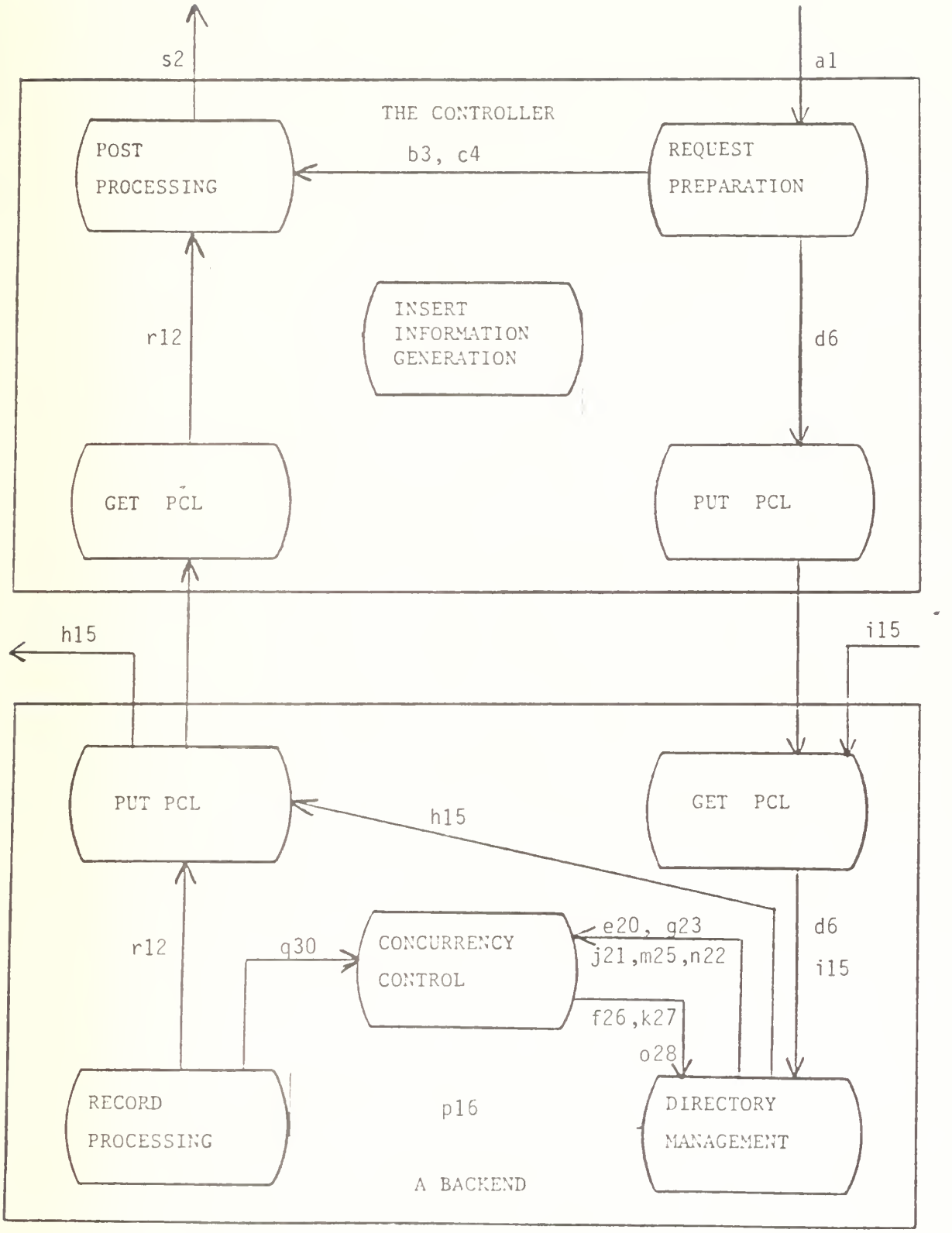


Figure 26. Sequence of Messages for a Retrieve Request with Aggregate Operations

then perform Cluster Search and signal CC to release the locks on the descriptor-id groups (m25). Next, DM will send the cluster ids for the retrieval to CC (n22). Once the cluster ids are locked, and the request can proceed with Address Generation and the rest of the request execution, CC signals DM (o28). DM will then perform Address Generation and send the retrieve request and the addresses to RECP (p16). Once the retrieval request has executed properly, RECP will tell CC that the request is done and the locks on the cluster ids can be release (q30). After the retrieval results have been aggregated in the backend, that result will be sent to PP for further aggregation (r12). When PP is done, the final results will be sent to the host (s2).

5.2.4. Sequence of Actions for an Update Request Causing a Change in Cluster

The sequence of actions for an update request that causes a record to change cluster is shown in Figure 27. This request is processed in two parts. First, after processing the update, it is determined that a record has changed cluster. Then, an insert is generated to actually store the new record. As in the previous examples, we will go through the complete execution of this request.

The host sends the update request to REQP (a1). REQP follows through by formatting the request and sending PP the number of requests in the transaction (b3). DM also receives a message from REQP, the parsed traffic unit (c6). DM sends the type-C attributes needed by the request to CC (d20). Once an attribute is locked and Descriptor Search can be performed, CC signals DM (e29). DM will then perform Descriptor Search and send a message to CC to release the lock on the attribute (f23). The DM in each backend will exchange descriptor ids with each of the other backends (g15 and h15). DM sends the descriptor-id groups needed by the update request to CC (i21). Once the descriptor-id groups are locked and Cluster Search can be performed, CC signals DM (j27). DM will then perform Cluster Search. DM will send the cluster ids to CC to check if the request can continue with Address Generation and the rest of request execution (k22). Once CC responds to DM that the request can go through (m28), DM will generate the disk addresses and send the request as well as the addresses to RECP(n16). When RECP retrieves the old values of the attribute being modified by the update, it will then send these old values and the new values to DM to check for records that have changed cluster (o19). A

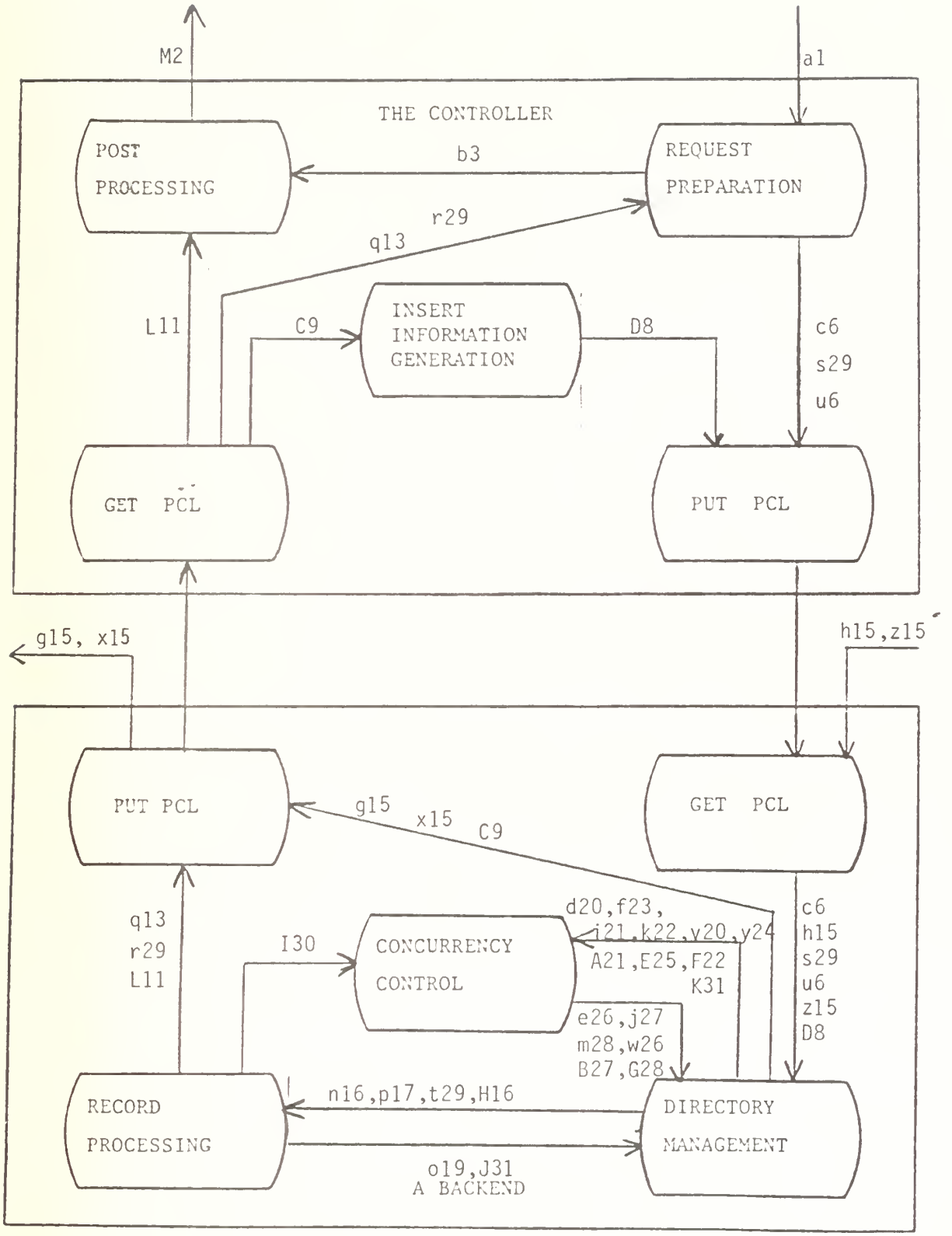


Figure 27. Sequence of Messages for an Update Request

reply will be sent to RECP from DM stating (for our example) that the update does cause a record to change cluster (p17). The change of cluster by a record requires an insert, therefore RECP will send the record that has changed cluster to REQP (q13). REQP will then generate an insert request. After sending all the updated records that have changed cluster to REQP, REQP sends a message to REQP (r29) indicating that there are no more changed-cluster records at this backend. (This message and the next message are needed to insure that the updated records are not inserted in the backends before the update requested has finished updating all the records. If the changed-cluster records are inserted too early, the update request may update some of them again.) After receiving the message (r29) from RECP in all the backends and after generating all the required insert requests, REQP sends a message to DM indicating that there will not be any more insert requests for this update request (s29). After receiving the message (s29) and performing directory-management processing for all the generated insert requests, DM sends a message to RECP indicating that there will not be any more insert request for this update request (t29). (RECP needs this message to determine when the update request is completely done. The update request is completely done when all the insert requests caused by it are done.)

Let us now describe how the generated insert is processed. The execution of this request proceeds as other insert requests. REQP sends DM the parsed traffic unit for the insert (u6). DM sends the type-C attributes needed by the insert request to CC (v20). Once an attribute is locked and Descriptor Search can be performed, CC signals DM (w26). DM will then perform Descriptor Search. From the DM, descriptor ids for the request will be sent to the other backends in the MDBS system (x15). DM also signals CC to release the locks on attributes (y24). The descriptor ids found by the other backends will be received by DM (z15). DM now sends the descriptor-id group for the request to CC (A21). (Note that we are now using capital letters for sequencing.) Once the descriptor-id group is locked and Cluster Search can be performed, CC signals DM (B27). DM will then perform Cluster Search. To determine where the insert will occur, DM will send the insert cluster id to IIG (C9). Once the backend has been selected, IIG will send the backend number to DM (D8). DM updates its directory tables if needed, and signals CC to release the lock held by the insert request on the descriptor-id group (E25). DM will send the insert cluster id to CC (F22). CC will respond to DM when the insert request

can proceed with Address Generation and the rest of the request execution (G28). With the go ahead from CC, DM will perform Address Generation and send RECP the request and its required disk address (H16). After the insert has occurred. RECP will notify CC that the insert request is done (I30).

After executing all the insert requests caused by the update request, RECP signals DM that the update request is completely done (J31). DM will free the space used by the update request and signal CC that the update request has finished (K31). RECP also sends the results of the update request to PP (L11) and PP notifies the host that the update has completed (M2).

6. CONCLUSIONS AND FUTURE PLANS

This report concludes the series of reports [Kerr82, He82, Boyn83] on the implementation of MDBS. We have finished the design, coding, implementation and testing of Version F which is the target version envisioned from the outset. Thus we can begin the next phases of experimentation, specifically, design verification and performance evaluation. To support these new phases of experimentation, we need to increase the number of backends in the system to, at least, six. The issues involving the hardware reconfiguration and expansion of MDBS are discussed in Section 6.1.

Additionally, we are investigating a security mechanism and language interfaces for relational and hierarchical data manipulation languages. These topics, along with the design verification and performance evaluation issues, are examined in Section 6.2. Finally, Section 6.3 contains a long-range goal of the Laboratory for Database Systems Research, i.e., the development of a general methodology for benchmarking database machines and database software systems.

6.1. Hardware Reconfiguration for MDBS

The current hardware configuration of MDBS consists of a VAX-11/780 running as the controller and two PDP-11/44s running as backends. Intercomputer communication is supported by three parallel communication links (PCL-11Bs). To increase the number of backends to six, we would need to purchase four more PDP-11/44s and four more PCL-11Bs. However, this has not been our original plan. Our plan was to use the Ethernet-like communications link and the smallest and cheapest minis which can support hard disks for our configuration. As always in the computer field, the intended hardware was not available in 1980. Since then we have used more powerful hardware, the PDP-11/44s and VAX-11/780 and more awkward hardware such as the PCL-11Bs. Thus, we are now investigating the possibility of replacing our current configuration with newer and more appropriate hardware. In particular, we are thinking of replacing our backends with the Digital LSI-11 series, either the 11/23, the 11/23+, or the 11/73. The gain is in cost and service. The cost of the LSI-11/23 or LSI-11/23+ is about half the cost of a PDP-11/44. The cost of the LSI-11/73 is about two-thirds the cost of a PDP-11/44. Since the software is portable, there is no problem in down-loading the existing software onto any of these three machines.

We are also considering a change in the communications hardware. When the implementation of MDBS began, the technology for local-area networks, e.g., Ethernet [Metc76], was not available. The replacement of the PCL hardware with an Ethernet or Ethernet-compatible network would standardize the communications hardware. Additionally, unlike Ethernet, the PCL is not a broadcast bus. We have required a broadcast bus for MDBS. In the current environment, when the controller needs to broadcast a message to all the backends, it must send the message separately to each backend. In other words, if there are two backends, the controller sends two messages. Thus, the message-passing overhead increases as the number of backends increases. An Ethernet will eliminate this overhead.

6.2. New Research

The new research on MDBS involves three major areas, a security mechanism, language interfaces to support the relational and hierarchical data manipulation languages, and the performance evaluation of the MDBS.

6.2.1. A Security Mechanism

Since security is an integral part of a database system, the design of a security mechanism is mandated. The design considerations of a security mechanism consists of two parts. First, the level(s), known as granule(s), at which the security control is applied must be determined. In MDBS there are four possibilities: the attribute level, the descriptor level, the cluster level, and the record level. Second, given the level(s) at which security is defined, the security mechanism is then specified. This is not an easy task, since the directory information about the levels changes dynamically when a new type-C descriptor or a new cluster is created.

6.2.2. Language Interfaces

There are three separate projects underway involving language interfaces. In designing language interfaces, we are providing the user access to MDBS using a variety of data manipulation languages. The series of papers [Bane77, Bane78, Bane80] demonstrated that a relational, hierarchical or network database can be transformed into an attribute-based database. Thus it is reasonable to design a language interface which maps a given data manipulation language into the attribute-based query language, so that the user may use the given language on the transformed database. One project involves the design

of a language interface for the relational query language, SQL [Astr76]. A second project involves the design of a language interface for DL/I of IMS [McGe77]. The third project is considering the various algorithms which can be used to implement the sort and join operations in the attribute-based system for the relational language interface.

6.2.3. Performance Evaluation

There are two projects dealing with the performance evaluation of MDBS. The internal performance evaluation project is measuring the execution times of the modules of the backend. These measurements include the time to process a particular message, the disk I/O time, the intra-computer and inter-computer message passing times, the process switch time, etc. The external performance evaluation project is measuring the throughput of the system. The throughput of MDBS is defined as the average number of user requests executed by the system in a second [Hsia81a]. The throughput of the system can be obtained for the four primary operations in MDBS.

6.3. What's Next

The work on performance evaluation and the relational and hierarchical language interfaces leads toward the ultimate goal of our research efforts: the specification of a general methodology to benchmark database machines and database software systems. We intend to extend the earlier work on benchmarking database machines and software systems which support the relational model [Stra84], to benchmark systems and machines based on the hierarchical and network data models.

Additionally, we intend to permit the comparison of two or more database systems. The benefit to such an approach is the ability to easily benchmark and compare similar and dissimilar database systems, i.e., a relative comparison. However, this approach does not preclude absolute comparison where only a single system has to be benchmarked. In this case, the benchmarks are of course written in the given data manipulation language.

REFERENCES

[Astr76] Astrahan, M. M., et al., "System R: a relational approach to data management," ACM Transactions on Database Systems 1:2, pp. 97-137.

[Bane77] Banerjee, J., Hsiao, D. K., and Kerr, D. S., "DBC Software Requirements for Supporting Network Databases," Technical Report, OSU-CISRC-TR-77-4, The Ohio State University, Columbus, Ohio, June 1977.

[Bane78] Banerjee, J., and Hsiao, D. K., "Concepts and Capabilities of a Database Computer," ACM Transactions on Database Systems, Vol. 4, No. 1, pp. 347-384, December 1978.

[Bane80] Banerjee, J., Hsiao, D. K., and Ng, F., "Database Transformation, Query Translation and Performance Analysis of a Database Computer in Supporting Hierarchical Database Management," IEEE Transactions on Software Engineering, March 1980.

[Boyn83] Boyne, R., et al., "The Implementation of a Multi-Backend Database System (MDBS): Part III - The Message-Oriented Version with Concurrency Control and Secondary-Memory-Based Directory Management," Technical Report, NPS-52-83-003, Naval Postgraduate School, Monterey, California, March 1983.

[Date83] Date, C. J., An Introduction to Database Systems, Volume II, Addison-Wesley, 1983.

[DEC79a] "PCL11-B Parallel Communication Link Differential TDM Bus," Digital Equipment Corp., Maynard, Mass., 1979.

[He82] He, X., et al., "The Implementation of a Multi-Backend Database System (MDBS): Part II - The First Prototype MDBS and the Software Engineering Experience," Technical Report, NPS-52-82-008, Naval Postgraduate School, Monterey, California, July 1982.

[Hsia81a] Hsiao, D.K. and Menon, M.J., "Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth (Part I)," Technical Report, OSU-CISRC-TR-81-7, The Ohio State University, Columbus, Ohio, July 1981.

[Hsia81b] Hsiao, D.K. and Menon, M.J., "Design and Analysis of a Multi-Backend Database System for performance Improvement, Functionality Expansion and Capacity Growth (Part II)," Technical Report, OSU-CISRC-TR-81-8, The Ohio State University, Columbus, Ohio, August 1981.

[Kerr82] Kerr, D.S., et al., "The Implementation of a Multi-Backend Database System (MDBS): Part I - Software Engineering Strategies and Efforts Towards a Prototype MDBS," Technical Report, OSU-CISRC-TR-82-1, The Ohio State University, Columbus, Ohio, January 1982.

[McGe77] McGee, W. C., "The IMS/VS System," IBM System Journal 16, No. 2, June 1977.

[Metc76] Metcalfe, R. M., and Boggs, D. R., "Ethernet:Distributed Packet Switching for Local Computer Networks," Communications of the ACM, vol. 19, pp. 395-404, July 1976.

[Stra84] Strawser, P. R., "A Methodology for Benchmarking Relational Database Machines," Ph. D. Dissertation, The Ohio State University, 1984.

[Ullm82] Ullman, J. D., Principles of Database Systems, Computer Science Press, 1982.

APPENDIX A

HOW TO READ AND FOLLOW THE PROGRAM SPECIFICATIONS

The appendices in this series have contained the detailed design of MDBS. In Appendix B, the programs for the directory management concurrency control are described and specified. In Appendix C, the programs for directory management using secondary memory are described and specified. These programs represent those parts of MDBS that have been newly designed and redesigned, since the first three reports in this series were written.

A.1 Parts within an Appendix

Each appendix begins with a introduction which outlines the major components of the design. For example, the design of the controller subsystem, presented in [He82], consisted of three major parts: request preparation, insert information generation and post processing. The design of a backend subsystem also consists of three major parts: directory management, record processing and concurrency control. Primary-memory-based directory management and record processing, were presented in the previous reports. The third part, namely, concurrency control, in presented in Appendix B of this report. Finally, the revisions for secondary-memory-based directory management are presented in Appendix C.

A.2 The Format of a Part

In each part, we provide the following documentation elements:

(1) Title of the part,

(2) Name of the design,

(3) Name of the designer,

(4) Date the design was first submitted,

(5) Dates of design modifications,

(6) Statements of the design purpose, and of the input and output requirements,

(7) Formal specifications of the input and output, if necessary,

(8) Procedure names used in the design,

(9) Jackson chart of the design, if necessary,

(10) Data structures used in the design,

(11) Program specification of the design.

A.3 Documentation Techniques for a Part

In the previous section, we listed the various documentation elements. They are used to describe a design. Documentation elements 1 through 5 are written in English phrases. Document element 6 is written in prose. On the other hand, document elements 7 through 11 can be expressed more effectively using other means. Specifically, we have used Backus-Naur form (BNF) for writing the specifications in document element 7.

The procedure names of document element 8 are shown in a program hierarchy. The use of the hierarchy makes clear the calling sequences of the procedures named. The data structures of documentation element 10 are specified in either the system specification language (SSL) or in the C programming language. In documentation element 11, the procedures, themselves, are specified in SSL.

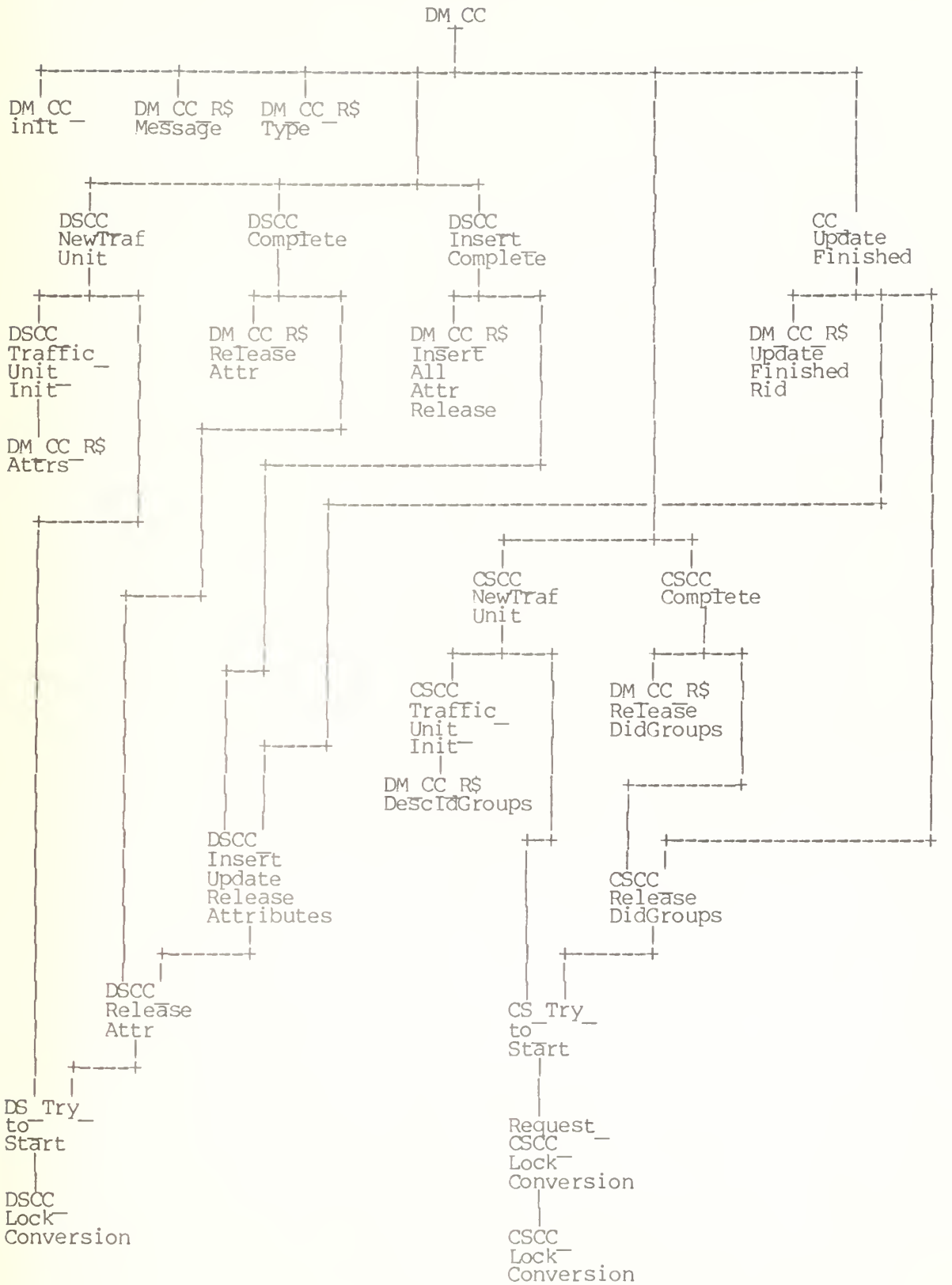
Except for the programming team that writes the procedures, other teams will usually not be interested in the internal logic of the procedures. Consequently, they need only know the higher-level specifications of the procedures. The SSL employed in MDBS is an ideal specification language for revealing the design of the procedures from a top-to-bottom-and-layer-to-layer way. It also works well with the hierarchical organization of procedures.

THE SSL SPECIFICATIONS FOR DIRECTORY MANAGEMENT CONCURRENCY CONTROL

The system specification for directory management concurrency control is given in this appendix.

```
/*    (1) Directory Management Concurrency Control                          */
/*    (2) Design:        : DM_CC                                            */
/*    (3) Designers      : A. Orooji, D. S. Kerr                            */
/*    (4) Date           : July 24, 1983                                    */
/*    (5) Modified       : December 28,1983                                 */
/*    (6) Purpose        :                                                  */
/*      The directory data, namely the descriptor-to-descriptor-id          */
/*  table(DDIT) and the cluster-definition table(CDT), may be               */
/*  modified during request execution.  Thus before descriptor search(DS)   */
/*  can access DDIT or cluster search(CS) can access CDT, appropriate       */
/*  locks must be obtained.                                                 */
/*      There are two types of locks: READ and WRITE.  A type-C attribute   */
/*  must be locked before  a request can perform DS on that attribute.      */
/*  To avoid deadlock the type-C attributes are sorted before being         */
/*  sent to DM_CC.                                                          */
/*      All descriptor-id-groups needed by a request must be locked before  */
/*  the request can perform CS.  Each descriptor-id group needed by a       */
/*  request is sorted (before being sent to DM_CC), and all the             */
/*  descriptor-id groups needed by a request are sorted.                    */
```

(8) Procedure Hierarchy for DM_CC

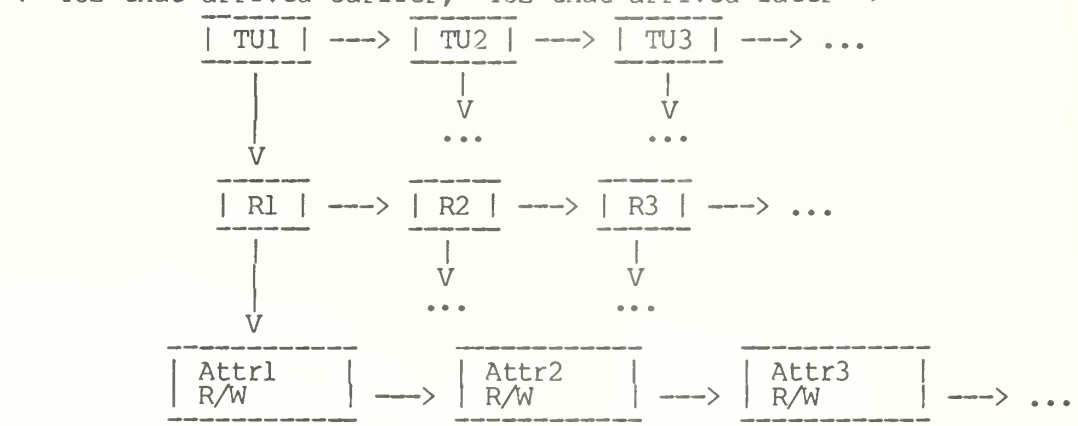


List of abbreviations:

ATUT	-	attribute-to-traffic-unit table
CS	-	cluster search
DM	-	directory management
DM_CC	-	directory management concurrency control
DS	-	descriptor search
TU	-	traffic unit
TUAT	-	traffic-unit-to-attribute table
TUDIGT	-	traffic-unit-to-descriptor-id-groups table

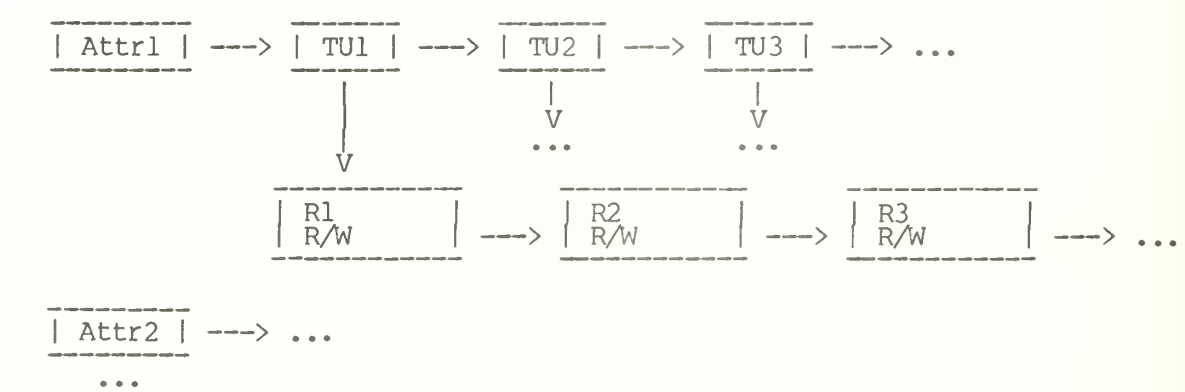
Traffic-Unit-to-Attribute Table (TUAT):

This table contains a list of traffic units and the type-C attributes needed by the requests in the traffic units. A type-C attribute needed by a request is locked before Descriptor Search (attribute-being-modified in an update request is also locked if it is type C).



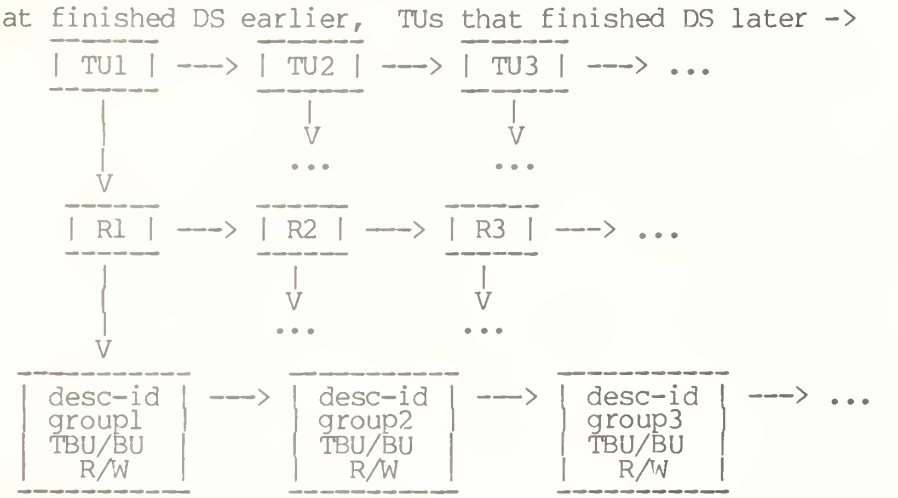
Attribute-to-Traffic-Unit Table (ATUT):

This table has the same information as TUAT, but it is based on attributes.



Traffic-Unit-Descriptor-Id-Groups Table (TUDIGT):

This table contains a list of traffic units and the descriptor-id groups needed by the requests in the traffic units. The descriptor-id groups needed by a request are locked before Cluster Search.



*/

(11) Program Specifications

```
1       task()  /* DM_CC (directory management concurrency control) */
2       {
           /* do initialization */
3          DM_CC_init();
4          while ( TRUE )
5             {
              / * get the next message for DM_CC */
6             DM_CC_R$Message();
              /* get the message type */
7             MsgType = DM_CC_R$Type();
8             switch ( MsgType )
9                {
10                case DS_NewTU:  /* a new traffic unit and the type-C
                                     attributes needed by the requests in it */
11                      DSCC_NewTrafUnit();
12                      break;

13                case CS_NewTU:  /* a new traffic unit and the descriptor-id
                                     groups needed by the requests in it */
14                   CSCC_NewTrafUnit();
15                   break;

16                case DS_ReleaseAttr:  /* a type-C attribute needed in DS is
                                           released */
17                   DSCC_Complete();
18                   break;

19                case DS_InsertReleaseAllAttrs: /* DS for insert releasing all
                                                    the type_C attributes */
20                   DSCC_InsertComplete();
21                   break;
```

```
22              case CS_ReleaseDidGroups:  /* the descriptor-id groups needed
                                              in CS are released */
23                 CSCC_Complete();
24                 break;

25              case UpdateFinished: /* An update is finished, so release
                                        update attribute(if any),
                                        descriptor-id groups and cluster */
26                 CC_UpdateFinished();
27                 break;


28           }/* end switch */

29        }/* end while */

30     }/* end main() */


26.1   CC_UpdateFinished()
          /* An update is finished, so release update attribute(if any), */
          /* descriptor-id groups and clusters                            */
26.2   {
          /* receive the request id */
26.3            DM_CC_R$UpdateFinishedRid( FinishedRid );

          /* find and remove update attribute, if any, from TUAT & ATUT */
26.4            DSCC_InsertUpdateReleaseAttributes( FinishedRid );

          /*  find and remove all descriptor-id groups from TUDIGT */
26.5            CSCC_ReleaseDidGroups( FinishedRid );

          /* find and remove all cluster-ids from TUCT and CTUT */
                /* (included here only for completeness, this is actually */
                /*  part of database concurrency control, NOT directory   */
                /*  management concurrency control) */

26.6   } /* end CC_UpdateFinished */


11.1   DSCC_NewTrafUnit()
          /* This routine is used when a new traffic unit and the type-C   */
          /* attributes needed by the requests in it are sent from DM to   */
          /* DM_CC.                                                        */
          /* This routine adds the traffic unit to TUAT and ATUT, and it   */
          /* tries to start DS.                                            */
11.2   {
          /* The message contains a traffic unit and the type-C attributes
             needed by the requests in it.  The message is
                 { [rid1,(attr1i, attr1j, ...)],
                   [rid2,(attr2i, attr2j, ...)],
                   ...                                 }     */

          /* Receive the traffic unit and the type-C attributes. */
          /* Enter the requests and the type-C attributes needed by them
             into TUAT and ATUT. */
11.3            DS_Traffic_Unit_Init( FirstRid , FirstAttribute );

          /* Try to convert locks on as many attributes as possible. */
11.4            DS_Try_to_Start( FirstRid , FirstAttribute );

11.5   }/* end DSCC_NewTrafUnit */
```

```
17.1    DSCC_Complete()
                /* a type-C attribute needed in DS is released */
17.2    {
              /* Receive the request id and the type-C attribute. */
17.3              DM_CC_R$ReleaseAttr( ReleasingRid, ReleasedAttr);
              /* Remove the attribute from TUAT and ATUT,
                        and try to start DS for other request(s). */
17.4              DSCC_ReleaseAttr( ReleasingRid, ReleasedAttr);

17.5    } /* end DSCC_Complete */


20.1    DSCC_InsertComplete()
              /* DS for insert is releasing all the type-C attributes */
20.2    {
            /* Receive the request id. */
20.3              DM_CC_R$InsertAllAttrRelease( ReleasingRid );
            /* Release the attributes. */
20.4              DSCC_InsertUpdateReleaseAttrs( ReleasingRid );

20.5    } /* end DSCC_InsertComplete */



11.3.1  DS_Traffic_Unit_Init( FirstRid , FirstAttr )
            /* Setup TUAT and ATUT entries for the new traffic unit. Return */
            /* pointers to the traffic unit, 1st request, and the position  */
            /* of the first attribute for that request.                     */
11.3.2  {
            /* Receive the traffic unit and the type-C attributes. */
11.3.3            DM_CC_R$Attrs( ... );

            /* Enter the requests and the type-C attributes needed by them into
               TUAT and ATUT. */
11.3.4            for each request in the traffic unit
11.3.5              {
11.3.6                for each attribute needed by this request
11.3.7                  {
11.3.8                    add an entry to TUAT;
                              /* the entry contains the following information
                                    attribute name or id
                                    mode (R/W)          */
11.3.9                    add an entry to ATUT;
                              /* the entry contains the following information
                                    rid
                                    mode (R/W)          */

11.3.10                 }/* end for */

11.3.11               }/* end for */

11.3.12     return first request id and first attribute

11.3.13 } /* end DS_Traffic_Unit_Init */
```

```
C=11.4 or B.8
C.1     DS_Try_to_Start( FirstRid , FirstAttribute )
          /* Try to convert locks on as many attributes as possible.  For each
             request in the traffic unit, we start with the first attribute
             needed by the request and continue with other attributes needed
             by the request.
C.2       {
C.3         for each request in the traffic unit
C.4           {
C.5             rid = request id of the request in the traffic unit;
C.6             TUAT_AttrPtr = pointer to the first attribute (TUAT entry) needed
                               by this request;
C.7             MoreAttrs = TRUE;
C.8             ConvertFlag = TRUE;
C.9             while ( MoreAttrs )
C.10              {
                  /* try to convert lock on the next attribute needed by this
                     request */
C.11              ConvertFlag = DSCC_LockConversion( rid, TUAT_AttrPtr);
C.12              if ConvertFlag
C.13                {
C.14                send a message to DM to start DS on the attribute
                        TUAT_AttrPtr->attr for the request rid;
C.15                  set TUAT_AttrPtr to point to the next attribute (TUAT entry)
                        needed by this request.  If it was last attribute for
                        this request, set MoreAttrs to FALSE to indicate there
                        are no more attributes to convert lock;

C.16                }/* end if */

C.17              }/* end while */

C.18          }/* end for */

C.19      }/* end DS_Try_to_Start */



B = A.5 or 17.4
B.1     DSCC_ReleaseAttr( ReleasingRid, ReleasedAttr)
          /* This routine is used when DM sends a message to DM_CC signaling
             that a request has finished DS on an attribute, and the attribute
             can now be released.
             This routine releases the attribute and tries to start DS for
             other requests that are waiting for the attribute. */
B.2       {
B.3         remove the attribute entry from TUAT and ATUT;
            /* try to start DS for other requests */
B.4         for each request following the request ReleasingRid in ATUT for the
                                                  attribute ReleasedAttr
B.5           {
B.6             rid = request id of the next request in ATUT for the attribute;
B.7             TUAT_AttrPtr = pointer to the attribute ReleasedAttr (TUAT entry)
                               which is needed by the request rid;
                /* Try to convert locks on as many attributes as possible.  We
                   start with the attribute ReleasedAttr which is needed by the
                   request rid and continue with other attributes needed by the
                   request rid.
                   We stop when we get to an attribute that we cannot lock. */
B.8                DS_Try_to_Start( rid, TUAT_AttrPtr );

B.9           }/* end for */

B.10    }/* end DSCC_ReleaseAttr */
```

```
A = 20.4 or 26.4
A.1     DSCC_InsertUpdateReleaseAttributes( FinishedRid )
          /* find and remove update attribute, if any, from TUAT & ATUT */
A.2     {
A.3         for each type-C attribute
A.4           {
                /* remove the attribute from TUAT & ATUT and try to start DS */
A.5               DSCC_ReleaseAttr( FinishedRid , Attribute );

A.6           } /* end for */

A.7     } /* end DSCC_InsertUpdateReleaseAttributes */


C.11.1  DSCC_LockConversion( rid, TUAT_AttrPtr )
          /* This routine tries to concert lock on attribute TUAT_AttrPtr->attr
          for request rid.  It returns TRUE if the lock is converted, FALSE
          otherwise. */
C.11.2  {
C.11.3      ATUT_ReqPtr = pointer to the request rid in ATUT for
                          attribute TUAT_AttrPtr->attr

C.11.4      if TUAT_AttrPtr->mode = W then
C.11.5        {/* the request is asking for write access;
                    the access can be granted only if
                        (1) the request is the first request on the list,
                                i.e., no other request is using the attribute
                     or (2) the request is the FIRST insert-caused-by-an-update
                            that is the first request on the list. */

C.11.6          if   first request on ATUT for attribute TUAT_AttrPtr->attr
C.11.7            {/* Case (1): access granted */
C.11.8               return(TRUE);
C.11.9            }

C.11.10         else if     FIRST insert-caused-by-update & update is first
C.11.11           {/* Case (2): access granted */
C.11.12              return(TRUE);
C.11.13           }

C.11.14         else
C.11.15           {/* access cannot be granted */
C.11.16              return(FALSE);

C.11.17           }
C.11.18       }/* end then part of if TUAT_AttrPtr->mode = W */
C.11.19     else
C.11.20       {/* the request is asking for read access; the access can be
                granted only if all the earlier requests on ATUT are also
                read accessing */

C.11.21         for all the earlier entries in ATUT for the attribute
                                    TUAT_AttrPtr->attr
C.11.22           {
C.11.23            ATUT_Ptr = pointer to the next entry in ATUT for the attribute
                                                          TUAT_AttrPtr->attr
C.11.24            if ATUT_Ptr->mode = W
C.11.25              {/* access cannot be granted */
C.11.26                 return(FALSE);
C.11.27              }/* end if */
C.11.28           }/* end for */

                /* all the earlier requests are also read accessing, so the
                   access can be granted */
C.11.29            return(TRUE);

C.11.30       }/* end else part */

C.11.31 }/* end DSCC_LockConversion */
```

```
14.1    CSCC_NewTrafUnit()
           /* This routine is used when a new traffic unit and the
              descriptor-id groups needed by the requests in it are
              sent from DM to DM_CC.   This routine adds the traffic
              unit to TUDIGT, and it tries to start CS. */
14.2    {
           /* Get the traffic unit and store it. */
14.3          CS_Traffic_Unit_Init( FirstRid );

           /* Try to convert locks on as many descriptor-id groups
                        as possible. */
14.4          CS_Try_to_Start( FirstRid );

14.5    }/* end CSCC_NewTrafUnit */


23.1    CSCC_Complete()
              /* the descriptor-id groups needed in CS are released */
23.2    {
           /* receive the request id */
23.3          DM_CC_R$ReleaseDidGroups( ReleasingRid);
           /* remove the descriptor-id groups from TUDIGT,
                     and try to start CS */
23.4          CSCC_ReleaseDidGroups( ReleasingRid);

23.5     }/* end CSCCcomplete */


14.3.1  CS_Traffic_Unit_Init( FirstRid )
           /* Get the traffic unit and store it. */
           /* The message contains a traffic unit and the descriptor-id groups
              needed by the requests in it.  The message is
                  { [rid1,(desc-id groupli, desc-id grouplj, ...)],
                    [rid2,(desc-id group2i, desc-id group2j, ...)],
                    ...                                          }     */

           /* receive the traffic unit and the descriptor-id groups */
14.3.2        DM_CC_R$DescIdGroups( ... );

           /* enter the requests and their corresponding descriptor-id groups
              into TUDIGT */
14.3.3     for each request in the traffic unit
14.3.4       {
14.3.5        for each descriptor-id group needed by this request
14.3.6          {
14.3.7             add an entry to TUDIGT;
                      /* the entry contains the following information
                           descriptor-id group
                           mode (R/W)
                           category (TBU)     */

14.3.8          }/* end for */

14.3.9       }/* end for */

14.3.10 } /* end CS_Traffic_Unit_Init */
```

```
E = 14.4 or D.4
E.1     CS_Try_to_Start( FirstRid )
          /* Try to convert locks on as many descriptor-id groups as possible
             for this and later traffic units.  For each request in a traffic
             unit, we start with the first descriptor-id group needed by the
             request and continue with other descriptor-id groups needed by
             the request.
             The Cluster Search for a request can proceed when all the
             descriptor-id groups needed by the request are locked. */
E.2     {
E.3        for each request in this or later traffic units
E.4          {
E.5            TUDIGT_ReqPtr = pointer to the next request (TUDIGT entry) in
                               the traffic unit;
               /* try to convert locks on as many descriptor-id groups (needed
                  by this request) as possible   */
E.6            ConvertFlag = Request_CSCC_LockConversion( TUDIGT_ReqPtr);
E.7            if ConvertFlag
E.8              {/* all descriptor-id groups needed by the request are locked */
E.9               send a message to DM to start CS for the request
                           TUDIGT_ReqPtr->rid;
E.10             }/* end if */

E.11         }/* end for */

E.12    }/* end CS_Try_to_Start */


D = 23.4 or 26.5
D.1     CSCC_ReleaseDidGroups( ReleasingRid )
          /* This routine is used when DM signals DM_CC that the descriptor-id
             groups corresponding to a request can be released.
             This routine releases the descriptor-id groups and tries to start
             CS for other requests that are waiting. */
D.2     {
D.3        remove TUDIGT entries corresponding to the descriptor-id groups for
                the request ReleasingRid;

           /* try to start CS for other requests */
D.4             CS_Try_to_Start( next later request );

D.5     }/* end CSCC_ReleaseDidGroups */


E.6.1   Request_CSCC_LockConversion( TUDIGT_ReqPtr )
          /* This routine tries to convert locks on descriptor-id groups
             corresponding to the request TUDIGT_ReqPtr.  It returns TRUE if
             all the descriptor-id groups needed by the request are locked,
             FALSE otherwise.  */
E.6.2   {
          /* Try to convert locks on all the descriptor-id groups needed by
             the request.  We stop when we get to a descriptor-id group
             that we cannot convert lock */
E.6.3     for each descriptor-id group needed by the request TUDIGT_ReqPtr
                until we cannot convert lock
E.6.4       {
E.6.5         TUDIGT_ReqDidGroupPtr = pointer to the next descriptor-id group
                    needed by the request TUDIGT_ReqPtr;
E.6.6         if TUDIGT_ReqDidGroupPtr->category is not BU
E.6.7           {/* try to convert lock */
E.6.8              ConvertFlag = CSCC_LockConversion( TUDIGT_ReqDidGroupPtr );

E.6.9           }/* end if TUDIGT_ReqDidGroupPtr->category is not BU */

E.6.10      }/* end for */

E.6.11    return( ConvertFlag );

E.6.12  }/* end Request_CSCC_LockConversion */
```

```
E.6.8.1 CSCC_LockConversion( DidGroupPtr )
          /* This routine tries to convert locks on descriptor-id group,
             DidGroupPtr corresponding to the request TUDIGT_ReqPtr.  It
             returns TRUE if all the descriptor-id groups needed by the
             request are locked, FALSE otherwise. */
E.6.8.2 {
          /* Try to convert locks on all the descriptor-id groups needed by
             the request.  We stop when we get to a descriptor-id group that
             we cannot convert lock */
E.6.8.3     for each descriptor-id group needed by the request TUDIGT_ReqPtr
E.6.8.4       {
E.6.8.5       TUDIGT_ReqDidGroupPtr = pointer to the next descriptor-id group
                                      needed by the request TUDIGT_ReqPtr;

E.6.8.6       if TUDIGT_ReqDidGroupPtr->category is not BU
E.6.8.7         {/* try to convert lock */
                 /* check this descriptor-id group with all the descriptor-id
                    groups corresponding to the EARLIER requests in TUDIGT */
E.6.8.8          for each descriptor-id group corresponding to EARLIER requests
E.6.8.9            {
E.6.8.10             TUDIGT_Ptr = pointer to TUDIGT entry for
                                  this descriptor-id group

                     /* we have to compare the two descriptor-id groups only if
                        one (or both) request is asking for write access (two
                        reads can go concurrently) */
E.6.8.11             if TUDIGT_ReqDidGroupPtr->mode = W or TUDIGT_Ptr->mode = W
E.6.8.12               {
E.6.8.13                if the two descriptor-id groups have conflicts,
                                i.e., there can be a descriptor-id group
                                     containing both groups
E.6.8.14                  {/* lock cannot be converted */
E.6.8.16                      return(FALSE);
E.6.8.17                  }/* end if */
E.6.8.18               }/* end if  */

E.6.8.19           }/* end for */

                   /* We have compared TUDIGT_ReqDidGroupPtr with all the earlier
                      descriptor-id groups, so the lock can be converted */
E.6.8.20            TUDIGT_ReqDidGroupPtr->category = BU;

E.6.8.21         }/* end if TUDIGT_ReqDidGroupPtr->category is not BU */

E.6.8.22       }/* end for */

             /* all the descriptor-id groups for the request have been locked */
E.6.8.23       return(TRUE);

E.6.8.24}/* end CS_CC_LockConversion */
```

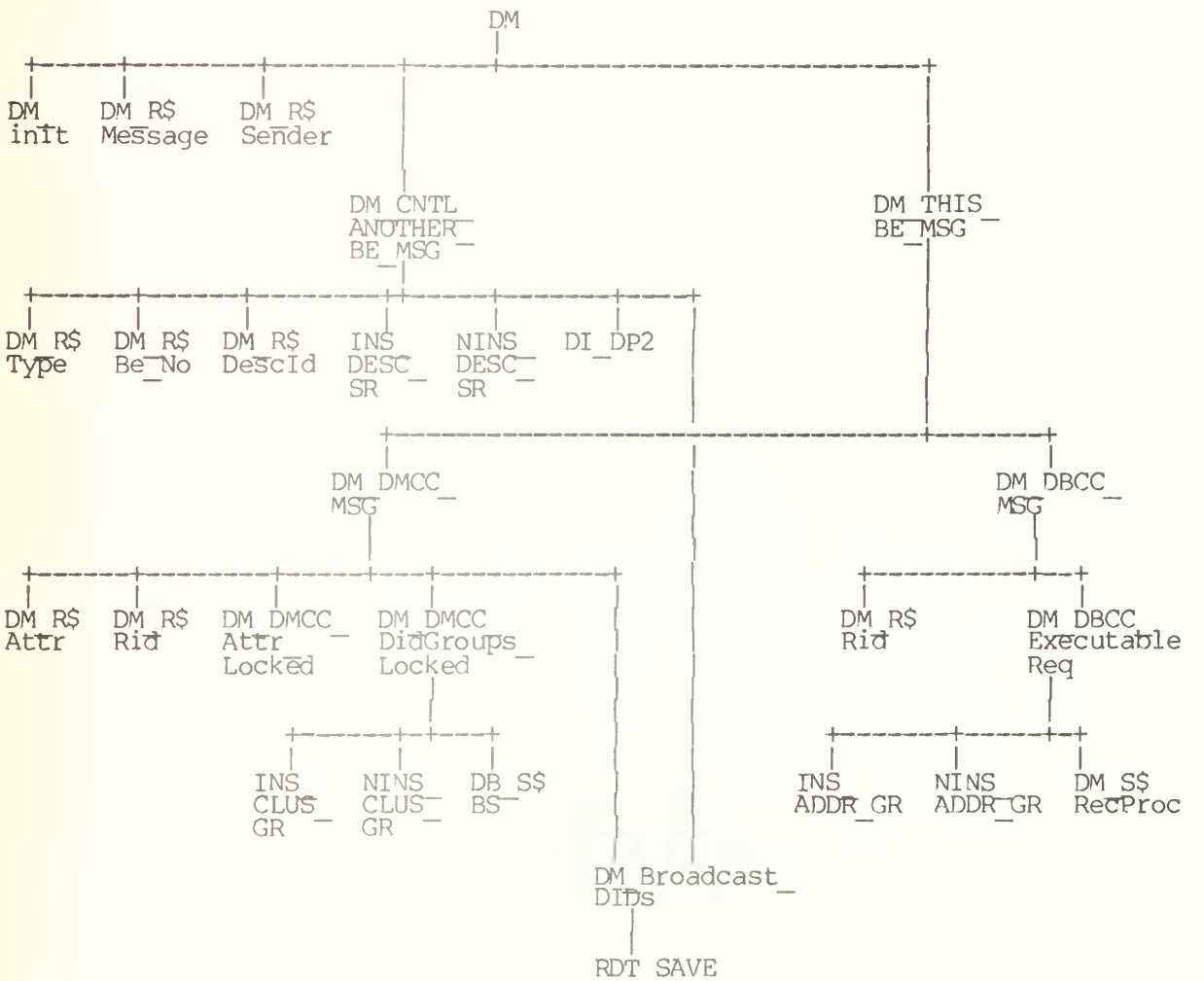
APPENDIX C

THE SSL SPECIFICATIONS FOR DIRECTORY MANAGEMENT

The system specification for directory management due to concurrency control of directory data is given in this appendix.

```
/*      (1) Directory Management with Concurrency Control on Directory Data */
/*      (2) Design:      : DM                                               */
/*      (3) Designer     : A. Orooji                                        */
/*      (4) Date         : July 7, 1983                                     */
/*      (6) Purpose      :                                                  */
/*        The directory data, namely the descriptor-to-descriptor-id        */
/* table(DDIT) and the cluster-definition table(CDT), may be                */
/* modified during request execution.  Thus before descriptor search(DS)    */
/* can access DDIT or cluster search(CS) can access CDT, appropriate        */
/* locks must be obtained.  Directory management was modified to allow      */
/* for concurrency control of this directory data.                          */
```

(8) Procedure Hierarchy for Directory Management(DM)



```
/*
   List of abbreviations:
      AG   -   address generation
      AT   -   attribute table
      CC   -   concurrency control
      CDT  -   cluster-definition table
      CS   -   cluster search
      DB   -   database
      DDIT -   descriptor-to-descriptor-id table
      DM   -   directory management
      DS   -   descriptor search
      IIG  -   insert information generation

*/



1       main()  /* Directory-Management */
2       {
           /* do initialization */
3          DM_init();
4          while ( TRUE )
5            {
              /* get the next message for Directory-Management */
6             DM_R$Message();
              /* get the sender name of the message */
7             sender = DM_R$Sender();
8             switch ( sender )
9               {
10               case G_PCLB: /* message from controller or another backend */
11                  DM_CNTL_ANOTHER_BE_MSG();
12                  break;

13               case THIS_BACKEND:  /* message from this backend */
14                  DM_THIS_BE_MSG(sender);
15                  break;

16               default:
17                  error;
18                  break;

19             }/* end switch */

20          }/* end while */

21      }/* end main */
```

```
11.1    DM_CNTL_ANOTHER_BE_MSG()
          /* This routine is used when there is a message for Directory    */
          /* Management from the controller or another backend.            */
11.2    {
          /* get the type of the message */
11.3      MsgType = DM_R$Type();
11.4      switch ( MsgType )
11.5        {
11.6        case ParsedTrafUnit:  /* requests from Request-Preparation */
              /* all the type-C attributes that are needed by the requests
                 in the traffic unit have to be sent to DS_CC together
                 (at once) */
11.7          send all the type-C attributes needed by the traffic unit
                                        to DS_CC;
                 /* the type-C attributes needed by the traffic unit:
                      . for an insert request in the traffic unit, all the
                        type-C attributes in the record (lock for write
                        access)
                      . for a non-insert request in the traffic unit, the
                        type-C attributes in that part of the query that this
                        backend will perform DS on (lock for read access)
                      . for an update request, the attribute-being-modified
                        if it is type-C (lock for write access) */
              /* process the requests one by one */
11.8          for each request in ParsedTrafUnit
11.9            {
11.10             if ( ReqType == INSERT )
11.11               {
11.12                 done = INS_DESC_SR( ... );
                             /* INS_DESC_SR returns true if it performs DS
                                on all the keywords that this backend is
                                supposed to perform DS on. */
                      /* Cluster Search & Address Generation are done later */
11.13               }
11.14             else
11.15               {/* request is non-insert */
11.16                 done = NINS_DESC_SR( ... );
                             /* NINS_DESC_SR returns true if it performs DS
                                on all the predicates that this backend is
                                supposed to perform DS on. */
                      /* Cluster Search & Address Generation are done later */
11.17               }
11.18             if ( done == true )
                    /* DS is done on all keywords/predicates that this
                       backend is supposed to do DS on;  broadcast the
                       descriptor ids to the other backends */
11.19               DM_Broadcast_DIDs( ... );
11.20           }/* end for */
11.21         break;

11.22       case NewDesc:  /* new descriptor from Descriptor-Id-Generator */
              /* receive the descriptor id generated in the controller */
11.23         DM_R$DescId(&rid, &predicate, &new_desc_id);
              /* store the descriptor id and indicate that DS is done for a
                 keyword */
11.24         done = DI_DP2(&rid, &predicate, &new_desc_id);
                /* DI_DP2 returns true if DS is done on all the keywords
                   that this backend is supposed to perform DS on. */
11.25         if ( done == true )
                /* DS is done on all keywords that this backend is supposed
                   to do DS on;  broadcast the descriptor ids to the other
                   backends */
11.26           DM_Broadcast_DIDs( ... );
11.27         break;
```

```
11.28          case BeNo:  /* backend number (selected for record insertion)
                              from Backend-Selector */
                  /* receive the backend number */
11.29             DM_R$Be_No(&rid, &be_no, &cid);
11.30             if this backend is supposed to insert the record
11.31               {
11.32                update directory tables if needed (new cluster);
11.33               }
11.34             release the descriptor-id group locked by the request;
11.35             if all the requests in the traffic unit have finished
                                 Cluster Search
11.36               {/* the traffic unit is sent to DB_CC when all the requests
                        in it have finished CS */
11.37                if all the earlier TUs which had conflict with this TU
                                 in CS have been sent to DB_CC
                        /* recall: traffic units are sent in order to DB_CC */
11.38                   send the traffic unit to DB_Concurrency-Control;
11.39               }
11.40             break;

11.41          case ...

11.42         }/* end switch */

11.43    }/* end DM_CNTL_ANOTHER_BE_MSG */


14.1     DM_THIS_BE_MSG(sender)
           /* This routine is used when there is a message for Directory
              Management from a task in the same backend. */
14.2     {
14.3        switch ( sender )
14.4          {
14.5          case DM_ConcurrencyControl:
14.6             DM_DMCC_MSG();
14.7             break;

14.8          case DB_ConcurrencyControl:
14.9             DM_DBCC_MSG();
14.10            break;

14.11         case RecordProcessing:
                 ...
14.12            break;

14.13         default:
14.14            error;
14.15            break;

14.16        }/* end switch */

14.17    }/* end DM_THIS_BE_MSG */
```

```
14.6.1   DM_DMCC_MSG()
           /* This routine is used when there is a message for Directory    */
           /* Management from DM_Concurrency Control (in the same backend). */
14.6.2   {
           /* get the type of the message */
14.6.3      MsgType = DM_R$Type();
14.6.4        switch (MsgType )
14.6.5        {
14.6.6         case AttrLocked:  /* attr needed by a request is locked */
                 /* get the attribute name */
14.6.7            DM_R$Attr(&rid, attr);
                 /* do the descriptor search if needed */
14.6.8            done = DM_DMCC_Attr_Locked(&rid, attr)
                        /* DM_DMCC_Attr_Locked returns true if DS is done on
                           all the keywords/predicates that this backend is
                           supposed to perform DS on. */
14.6.9            if ( done == true )
                    /* DS is done on all keywords/predicates that this backend
                       is supposed to do DS on;  broadcast the descriptor ids
                       to the other backends */
14.6.10               DM_Broadcast_DIDs( ... );
14.6.11           break;

14.6.12        case DidGroupsLocked:  /* descriptor-id groups needed by a request
                                         are locked */
                 /* receive the request id */
14.6.13           DM_R$Rid(&rid);
                 /* do the Cluster Search */
14.6.14           DM_DMCC_DidGroups_Locked(&rid);
14.6.15           break;

14.6.16        case ...

14.6.17       }/* end switch */

14.6.18  }/* end DM_DMCC_MSG */


14.9.1   DM_DBCC_MSG()
           /* This routine is used when there is a message for Directory    */
           /* Management from DB_Concurrency Control (in the same backend). */
14.9.2   {
           /* get the type of the message */
14.9.3      MsgType = DM_R$Type();
14.9.4      switch ( MsgType )
14.9.5        {
14.9.6         case ExecutableReq:  /* DB_CC has given permission to execute
                                       request, i.e., the request can continue with Address
                                       Generation and the rest of request execution */
                 /* receive the request id */
14.9.7            DM_R$Rid(&rid);
                 /* do the Address Generation */
14.9.8            DM_DBCC_ExecutableReq(&rid);
14.9.9            break;

14.9.10        case ...

14.9.11       }/* end switch */

14.9.12  }/* end DM_DBCC_MSG */
```

```
11.12.1 INS_DESC_SR( ... )
          /* This routine is used for processing insert requests.  It finds  */
          /* the descriptors that satisfy the keywords in an insert request. */
11.12.2 {
          /* if there are X keywords (in the record being inserted) and N   */
          /* backends, each backend performs descriptor search on X/N       */
          /* keywords                                                       */
11.12.3   for each keyword that this backend is supposed to do descriptor
                                search
11.12.4     {
11.12.5      if attr in keyword is not type C
11.12.6         do the descriptor search;
                   /* type-C attributes should be locked for write before DS */
11.12.7     }/* end for */

11.12.8   if DS is done on all keywords that this backend is supposed to do
                                DS on
11.12.9      return(true);
11.12.10  else
11.12.11     return(false);

11.12.12 }/* end INS_DESC_SR */


11.16.1 NINS_DESC_SR( ... )
          /* This routine is used for processing non-insert requests.  It  */
          /* finds the descriptors that satisfy the predicates in a         */
          /* non-insert request.                                            */
11.16.2 {
          /* if there are X predicates (in the query) and N backends, each */
          /* backend performs descriptor search on X/N predicates           */
11.16.3   for each predicate that this backend is supposed to do descriptor
                                search
11.16.4     {
11.16.5      if attr in predicate is not type C
11.16.6         do the descriptor search;
                   /* type-C attributes should be locked for read before DS */
11.16.7     }/* end for */

11.16.8   if DS is done on all predicates that this backend is supposed to do
                          DS on
11.16.9      return(true);
11.16.10  else
11.16.11     return(false);

11.16.12 }/* end NINS_DESC_SR */


11.24.1 DI_DP2(rid, predicate, new_desc_id)
          /* This routine is used for insert requests.  It is called when   */
          /* Insert-Information-Generation sends a new descriptor id to the */
          /* backends.  This routine adds the descriptor id to DDIT.  It    */
          /* also indicates that the descriptor id is ready for the insert  */
          /* request which is waiting for the descriptor id.                */
11.24.2 {
11.24.3   update the DDIT (and AT if needed);
11.24.4   for the insert request which is waiting for this descriptor id,
                indicate the descriptor search for that keyword is finished;
11.24.5   if DS is done on all keywords that this backend is supposed to do
                DS on
11.24.6      return(true);
11.24.7   else
11.24.8      return(false);

          /* the type-C attributes will be released before Cluster Search */

11.24.9 }/* end DI_DP2 */
```

```
14.6.8.1 DM_DMCC_Attr_Locked(rid, attr)
          /* This routine is used when DM Concurrency Control signals    */
          /* Directory Management that an attribute needed by a request  */
          /* is locked.                                                  */
14.6.8.2 {
14.6.8.3   ReqType = type of the request;
14.6.8.4   switch ( ReqType )
14.6.8.5     {
14.6.8.6      case INSERT:
                /* we recall that this backend locks all type-C attributes in
                   the record even those that other backends will do DS on, so
                   we need to check to see if this backend is supposed to do
                   DS;  we can have DM Concurrency Control not send a message
                   back for the attributes that will be processed in the other
                   backends [Directory Management, of course, must tell
                   DM Concurrency Control (when sending attributes) about this] */
14.6.8.7          if this backend is supposed to do descriptor search
14.6.8.8            {
14.6.8.9             do descriptor search for the keyword having the attribute;
                       /* if the type-C attribute has a new value, we need a new
                          descriptor.  In descriptor search, if no descriptor is
                          found, a message is sent to the Insert-Information-
                          Generation to generate a new descriptor id.        */

14.6.8.10            if descriptor search is done on all keywords
14.6.8.11               return(true);
14.6.8.12            else
14.6.8.13               return(false);
14.6.8.14           }/* end if */
                  /* the type-C attributes will be released before
                                  cluster search */
14.6.8.15         break;

14.6.8.16     case RETRIEVE:
14.6.8.17     case DELETE:
                /* we recall that, for non-insert requests, only the attributes
                   that this backend is supposed to do descriptor search on
                   are sent to DM Concurrency Control */
14.6.8.18         do the descriptor search for all predicates having
                                  the attribute;
14.6.8.19         release the attribute;
14.6.8.20         if descriptor search is done on all predicates
14.6.8.21            return(true);
14.6.8.22         else
14.6.8.23            return(false);
14.6.8.24         break;

14.6.8.25     case UPDATE:
14.6.8.26         do the descriptor search for all predicates having
                                  the attribute;
14.6.8.27         if attr is not the attribute-being-modified
14.6.8.28            release the attribute since DDIT will not be modified;
14.6.8.29         else
14.6.8.30            don't release the attribute yet since DDIT may be modified
                                  as a result of inserts caused by the update;
14.6.8.31         if descriptor search is done on all predicates
14.6.8.32            return(true);
14.6.8.33         else
14.6.8.34            return(false);
14.6.8.35         break;

14.6.8.36     default:
14.6.8.37         error;
14.6.8.38         break;

14.6.8.39    }/* end switch */

14.6.8.40 }/* end DM_DMCC_Attr_Locked */
```

```
14.6.14.1 DM_DMCC_DidGroups_Locked(rid)
          /* This routine is used when DM_Concurrency_Control signals      */
          /* Directory Management that the descriptor-id groups needed by a */
          /* request are locked.  The request can then proceed with        */
          /* Cluster Search.                                               */
14.6.14.2 {
14.6.14.3    ReqType = type of the request;
14.6.14.4    switch ( ReqType )
14.6.14.5      {
14.6.14.6       case INSERT:
                 /* do Cluster Search (i.e., find the id of the cluster to  */
                 /* which the record being inserted belongs)               */
14.6.14.7          cid = INS_CLUS_GR( ... );
                       /* the descriptor-id group locked by the request will be
                          released when cluster search is done and Insert-
                          Information-Generation has determined whether or not
                          there is a new cluster and CDT has been modified */
                 /* send the cluster id to Backend-Selector */
14.6.14.8          DM_SSBS( ... );
                     /* traffic unit will be sent to DB_Concurrency-Control
                        after Insert-Information-Generation responds */
14.6.14.9          break;

14.6.14.10      case RETRIEVE:
14.6.14.11      case DELETE:
                 /* do Cluster Search (i.e., find the ids of the clusters */
                 /* that satisfy the query in the request)                */
14.6.14.12         NINS_CLUS_GR( ... );
14.6.14.13         release the descriptor-id groups locked by the request;
14.6.14.14         if all the requests in the traffic unit have finished
                                Cluster Search
14.6.14.15           {/* the traffic unit is sent to DB_CC when all the requests
                                in it have finished CS */
14.6.14.16            if all the earlier TUs which had conflict with this TU
                                in CS have been sent to DB_CC
                       /* recall: traffic units are sent in order to DB_CC */
14.6.14.17               send the traffic unit to DB_Concurrency-Control;
14.6.14.18           }
14.6.14.19         break;

14.6.14.20      case UPDATE:
                 /* do Cluster Search (i.e., find the ids of the clusters
                 /* that satisfy the query in the request)                */
14.6.14.21         NINS_CLUS_GR( ... );
                       /* descriptor-id groups locked by the request will be
                          released after the update is completely done */
14.6.14.22         if all the requests in the traffic unit have finished Cluster
                                Search
14.6.14.23           {/* the traffic unit is sent to DB_CC when all the requests
                                in it have finished CS */
14.6.14.24            if all the earlier TUs which had conflict with this TU in
                                CS have been sent to DB_CC
                       /* recall: traffic units are sent in order to DB_CC */
14.6.14.25               send the traffic unit to DB_Concurrency-Control;
14.6.14.26           }
14.6.14.27         break;

14.6.14.28      default:
14.6.14.29         error;
14.6.14.30         break;

14.6.14.31     }/* end switch */

14.6.14.32 }/* end DM_DMCC_DidGroups_Locked */
```

```
14.9.8.1 DM_DBCC_ExecutableReq(rid)
           /* This routine is used when DB Concurrency Control signals  */
           /* Directory Management that a request can proceed with       */
           /* Address Generation and the rest of request execution.      */
14.9.8.2 {
14.9.8.3    ReqType = type of the request;
            /* perform the Address Generation for the request */
14.9.8.4    if ( ReqType == INSERT )
14.9.8.5      {/* insert request */
14.9.8.6       INS_ADDR_GR( ... );
14.9.8.7      }
14.9.8.8    else
14.9.8.9      {/* non-insert request */
14.9.8.10      NINS_ADDR_GR( ... );
14.9.8.11     }

            /* send the request to Record Processing */
14.9.8.12   DM_S$RecProc( ... );

14.9.8.13 }/* end DM_DBCC_ExecutableReq */


A = 11.19 or 11.26 or 14.6.10
A.1     DM_Broadcast_DIDs(rid)
           /* This routine broadcasts the descriptor ids found by this backend */
           /* to the other backends.  It also saves these descriptor ids       */
           /* [which are in request-descriptor table (RDT)] to be used in      */
           /* cluster search later.                                            */
A.2     {
A.3        find the RDT for the request;
           /* save the RDT */
A.4        RDT_SAVE(rid, RDT);

A.5        broadcast the descriptor ids to the other backends;

A.6     }/* end DM_Broadcast_DIDs */
```

```
A.4.1   RDT_SAVE(rid, RDT)
           /* This routine saves the descriptor ids [which are in request-  */
           /* descriptor table (RDT)] found by a backend (during Descriptor */
           /* Search).                                                      */
           /* This routine is called when this backend has finished DS or   */
           /* another backend has sent the descriptor ids that it has found */
           /* in DS.                                                        */
           /* When the descriptor ids found by all the backends have been   */
           /* received and saved, we can proceed to Cluster Search.         */
A.4.2   {
A.4.3      save the RDT;
           /* check to see if all the backends have sent the descriptor ids */
A.4.4      if all RDTs for the request have been saved
A.4.5        {/* the request has finished DS */
A.4.6         if the request is insert
A.4.7           {
A.4.8            release all the type-C attributes locked by the request;
                       /* we recall that for non-insert requests, each type-C
                          attribute is released immediately after DS is done
                          on that attribute */
A.4.9           }
A.4.10        if all requests in traffic unit have finished Descriptor Search
A.4.11          {/* the traffic unit is sent to CS_CC when all the requests
                    in it have finished DS */
A.4.12           if all the earlier TUs which had conflict with this TU in DS
                                                        have been sent to CS_CC
A.4.13             {
                    /* recall that traffic units are sent in order to CS_CC */
A.4.14              send the traffic unit to CS_Concurrency-Control;
                      /* the descriptor-id groups corresponding to the requests
                         in the traffic unit are sent to DM_Concurrency-Control
                         (for read/write access depending on the request types);
                         DM_Concurrency-Control will respond when all the
                         descriptor-id groups for a request have been locked.  At
                         that time, Cluster Search for that request can be
                         performed */
A.4.15             }

A.4.16          }/* end if all the requests in the TU have finished DS */

A.4.17       }/* end if all the RDTs for the request have been saved */

A.4.18  }/* end RDT_SAVE */
```

APPENDIX D : REMAINING ALGORITHMS FOR THE SECONDARY-MEMORY-BASED DIRECTORY MANAGEMENT

This appendix contains descriptions of the remaining algorithms for the secondary-memory-based directory management. The first section examines what happens when the directory data is modified. The second section explains how to determine if an updated record has changed cluster.

D.1 Updating the Directory Data

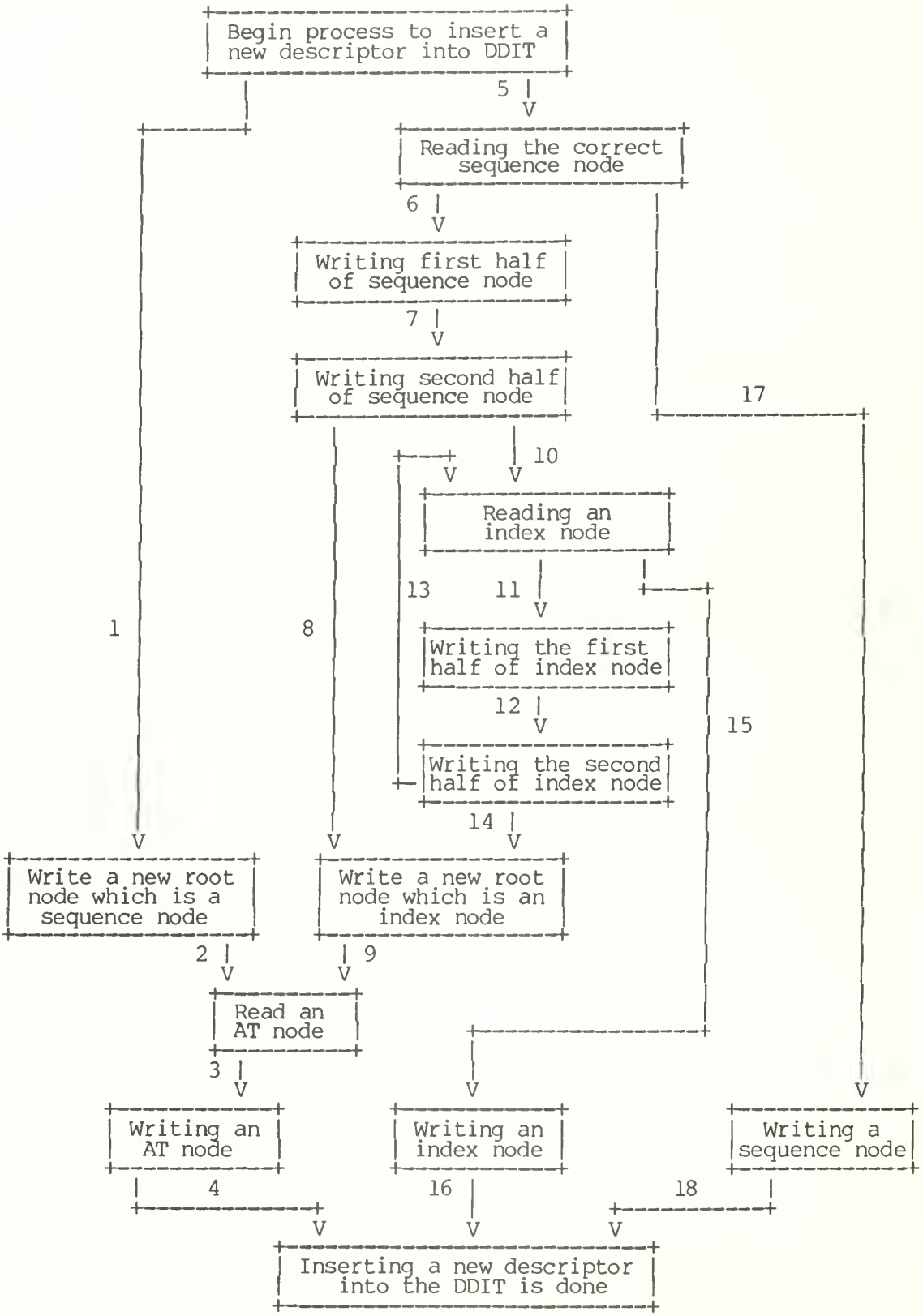
In this section, we explain the algorithms used to update the directory data. We examine two types of updates: one is due to the fact that a new type-C sub-descriptor is defined, and the other when a new cluster is defined. When a new type-C sub-descriptor is defined, the DDIT and the DCBMT must be modified. This is covered in Section D.1.1. When a new cluster is defined, the DCBMT is modified. We review this procedure in Section D.1.2.

D.1.1 Updating the DDIT and the DCBMT when a New Descriptor is Defined

As was described in Section 4.2, new descriptors are not inserted until descriptor search is finished for all the descriptors in the request. At that time, both the descriptor-to-descriptor-id table(DDIT) and the descriptor-id-cluster-bit-map table(DCBMT) must be updated. This processing is described in the next two sections.

(A) Updating the DDIT

The states and transitions for inserting a new descriptor into the DDIT are shown in Figure D.1. The descriptor information added to the DDIT consists of a descriptor id and a range of values which specify the new descriptor. The first step in inserting a new descriptor is to read the appropriate sequence node(5). (We recall that this sequence node was identified in the descriptor search phase. When the descriptor search for this descriptor was unsuccessful, the pointer to the sequence node where the new descriptor information should be inserted was then determined.) If there is room in this node, then the descriptor is inserted(17) and processing is finished(18). If there is no room in this node, the node must be split and both halves written to



Note : All reads and writes in the state diagram are for the DDIT unless otherwise specified.

Figure D.1. Inserting a New Descriptor into the DDIT

disk(6,7). Then a new entry must be added to the parent index node, which must first be read(1). If the parent index node is also full, then it will also have to be split and an entry added to its parent. In general, it may be necessary to add a new entry and split several parent index nodes(11,12,13). Once an index node is found that is not full, the last insertion is made(15) and processing terminates for that descriptor(16). In the worst case, the root node will be full. This occurs when the root node is either an index or a sequence node. In either case, the root node must be split and a new root node created(14 for index, 8 for sequence). In addition, since the attribute table contained a pointer to the old root node, the attribute table must also be modified(9,3). Then the processing is finished(4). This procedure is used when the DDIT exists for the given attribute.

There is a special case to consider when the DDIT does not exist for the given attribute. In this case, we are inserting the first descriptor for the given attribute. When the first descriptor for an attribute is inserted, it is only necessary to create the root node(1), which is a sequence node, and link the attribute table to it as before(2,3). This completes the processing for this case(4). After the descriptor has been added to DDIT, we must add the descriptor to the DCBMT.

(B) Updating the DCBMT

The states and transitions for modifying the DCBMT when a new descriptor is defined are given in Figure D.2. The bit map being added for a descriptor specifies which cluster(s) contain the descriptor. To find the correct place to insert the new descriptor into the DCBMT, we work with the descriptor id. Recall that descriptor ids are in the form (attribute id, descriptor-within-attribute). This pair is converted into a single descriptor number. These numbers are then subdivided into groups. For each group there is a bit-map set. When processing a new descriptor id, there may already be a bit-map set for the group which contains the new descriptor id.. If the bit-map set exists, it must be read(8) and updated(9). If the bit-map set does not exist, a bit-map set must be created(3). In either case, the bit map itself must be created. This bit map is subdivided into several blocks, each of which must be initialized and stored on the secondary memory(5). The last bit-map block is special(6) because after it is written, the insertion of this descriptor is

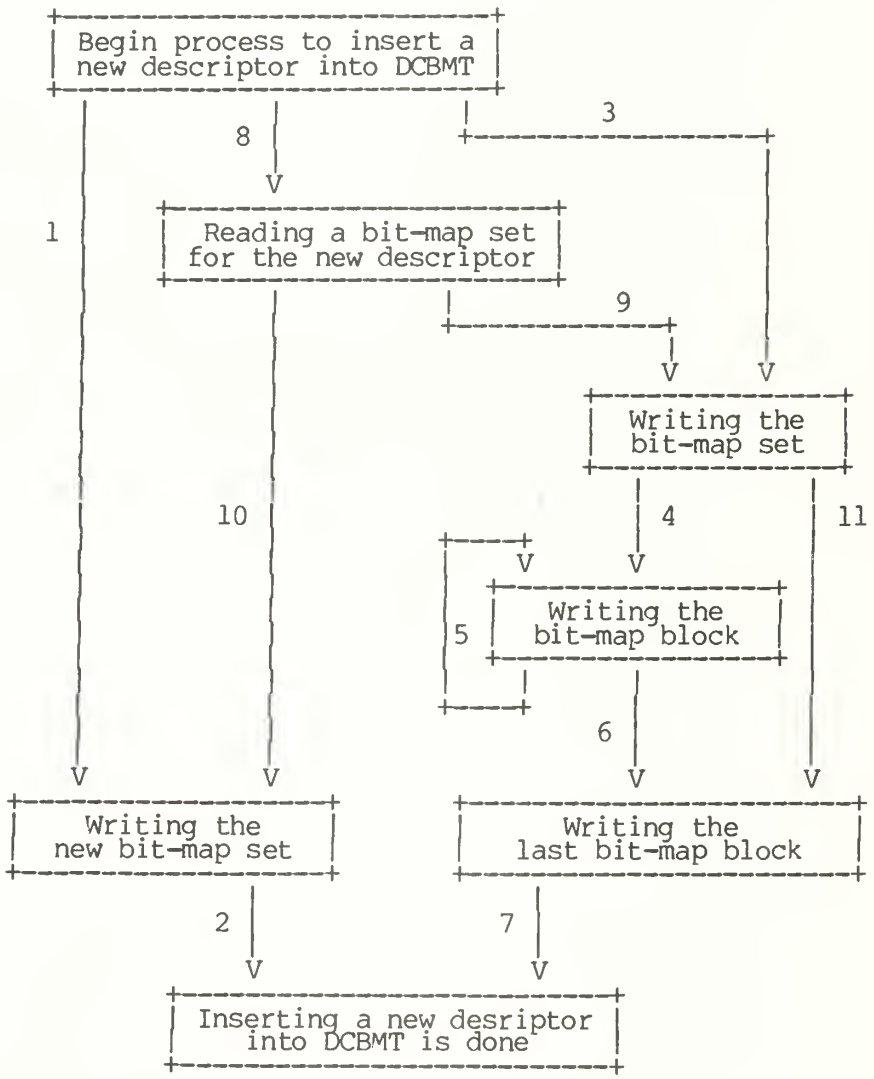


Figure D.2. Inserting a New Descriptor into the DCBMT

finished(7).

Two special cases must also be considered. First there may not yet be any database and therefore there will be no clusters. In this case, the bit-map set may or may not already exist. If the bit-map set does not exist, one is created(1). If the bit-map set does exist, it must be read(8) and updated(10). In either event, since there are no bit-map blocks for the new bit-map set, processing is done(2). The second special case occurs when there are only a few clusters so that there is only one bit-map block for each descriptor. In this case, after the bit-map set is read(8) and updated(9), there is only one bit-map block to be written(11), which completes the processing(7).

D.1.2 Updating the DCBMT when a New Cluster is Defined

When a new cluster is created by an insert request, it is necessary to add the new cluster id and corresponding descriptor-id set to the DCBMT. This addition occurs after the backend number at which the insert request will be executed is received from the controller. At this time a new column, corresponding to the new cluster id, must be added to the DCBMT. Thus a 1 bit has to be added to the bit map for each descriptor id in the descriptor-id set corresponding to the cluster. The bit maps for the other descriptor ids do not have to be modified, since all yet to be used entries were set to 0 when those bit maps were created.

Figure D.3 shows the states and transitions needed to insert the entry for one descriptor id into the DCBMT. These steps must be repeated for each descriptor id in the descriptor-id set corresponding to the new cluster. The first step is to read the bit-map set corresponding to the descriptor id(1). Then the first block of the bit map is read(5), followed by the rest of the bit-map blocks(6). Now there are two cases depending on whether or not there is space in the last bit-map block. If there is space in the last bit-map block, then the appropriate bit is changed to 1 and this block is written(9). On the other hand, if there is not space in the last bit-map block, then a new block must be created and linked to the previous last block. Thus a new secondary storage address is obtained and this address is put in the previous last block, which is then written(7). Then the new block is written(8). In either case, processing is finished for this descriptor id after the last

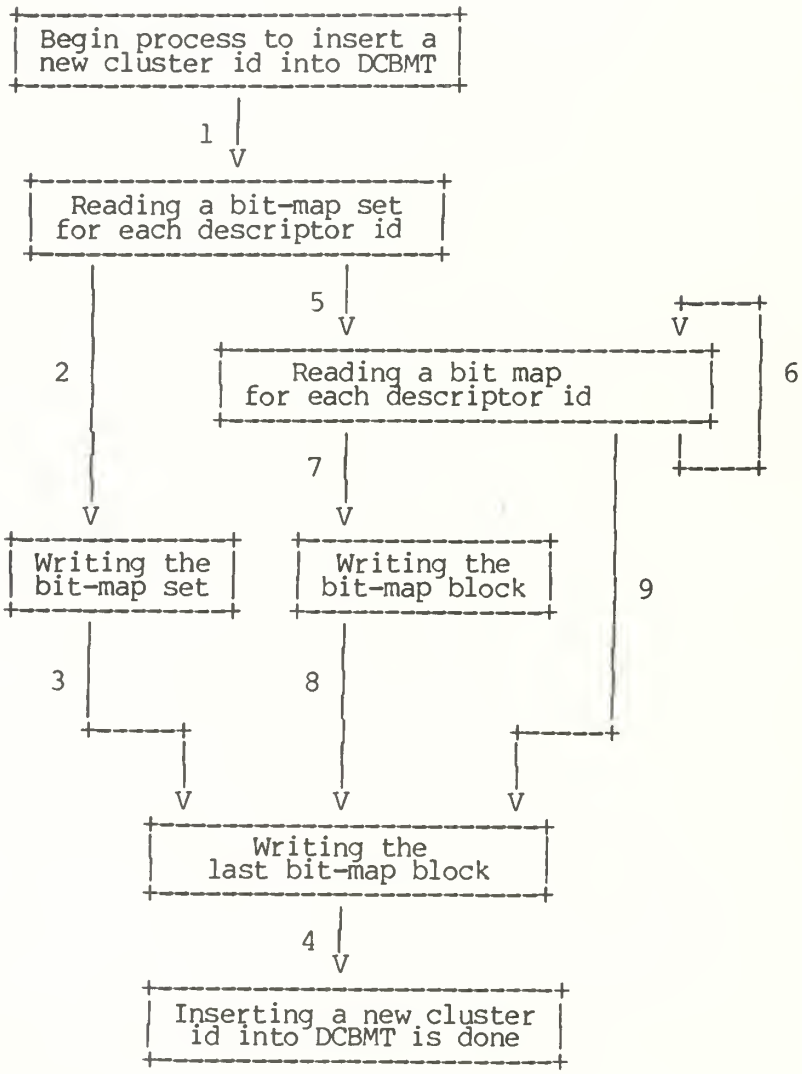


Figure D.3. Inserting a New Cluster Id into the DCBMT

bit-map block has been written(4). There is also one special case to consider. The cluster being added may be the first cluster in the database. In this case there will be no bit-map blocks. Thus the first bit-map block is created and linked to the bit-map set. The updated bit-map set is written(2) and so is the bit-map block(3), finishing the processing(4).

The above procedure is repeated for each descriptor id in the descriptor-id set corresponding to the cluster being defined. When this has been done, the DCBMT has been updated to contain the information for the new cluster.

D.2 Determining if an Updated Record Has Changed Cluster

When record processing updates a record, it must find out if the record has changed cluster. If it has, then an insert request must be generated so that the record can be inserted into the correct cluster. Otherwise, the record is updated in place. Record processing must send a message to directory management asking if the record has changed cluster. Directory management then must reply yes or no. If the attribute being modified is not a directory attribute, then it is clear that the record has not changed cluster. In addition, if the attribute being modified is a type-C attribute, then the record has changed cluster only if the old and new values are different. If they are the same, then the record has not changed cluster. In either case, it is not necessary to consult the directory data to determine if the record has changed cluster.

In the event the attribute being modified is a directory attribute which is not of type C, processing is more complex. In this case we must determine if the old and new values are derivable from the same descriptor id. Thus we must determine the descriptor ids for the old and new values. This determination requires searching the DDIT twice, once for each value.

The states and transitions to determine if an updated record has changed cluster is shown in Figure D.4 The first step in determining the descriptor id corresponding to the old value is to read the root node(1). Then other nodes of DDIT are read(2) until the appropriate sequence node is found. After the descriptor id corresponding to the old value has been determined, the search for the descriptor id corresponding to the new value begins by reading the

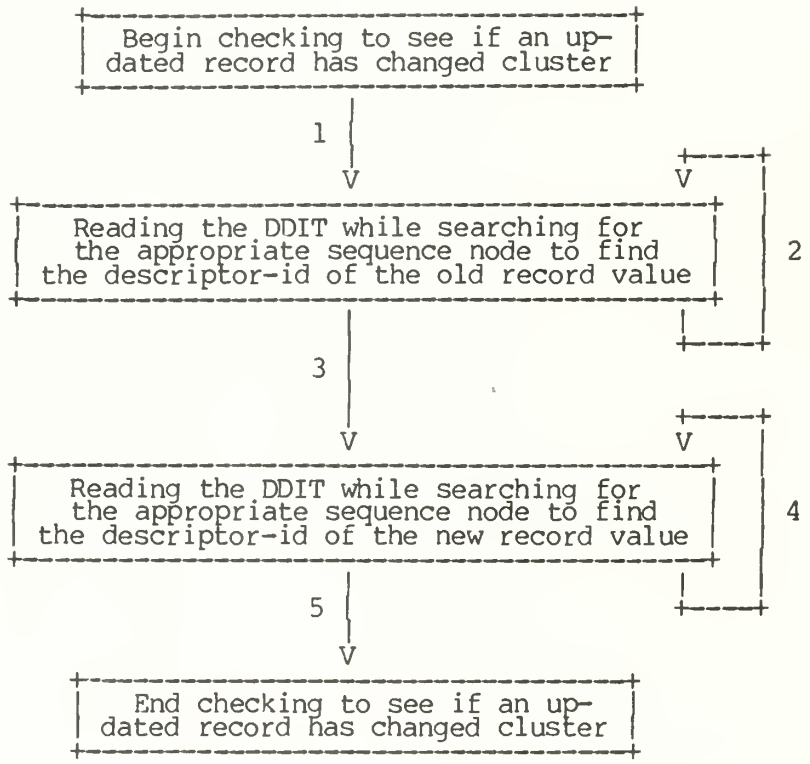


Figure D.4. Determining if an Updated Record Has Changed Cluster

root node again(3). Then other DDIT nodes are read(4) until the sequence node containing the descriptor id corresponding to the new value is found. After determining if the old and new descriptor ids are the same, a message is sent to record processing saying whether or not the record has changed cluster, completing the processing of this changed-cluster message(5).

One update request may cause several database records to be updated. Record processing must determine if each has changed cluster. Thus one update request may cause several messages asking if a record has changed cluster to be sent to the directory management. Only one of these is processed at a time. Others must wait. Thus after the directory management has determined whether one record has changed cluster, it should answer the same question for the next record, if any, that is waiting.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93943	1
Chairman, Code 52Hq Computer Science Department Naval Postgraduate School Monterey, CA 93943	194
Chief of Naval Research Arlington, VA 22217	1